

341

7P312C

214

C++ 设计新思维

泛型编程与设计模式之应用

Modern C++ Design

Generic Programming
and Design Patterns Applied



A1017658

Andrei Alexandrescu 著

侯捷 於春景 译

C++设计新思维

Modern C++ Design

Andrei Alexandrescu

Copyright ©2001 by Addison Wesley.

Simplified Chinese Copyright 2003 by Huazhong Science and Technology University Press and Pearson Education North Asia Limited.

All rights Reserved.

Published by arrangement with Pearson Education North Asia Limited, a Pearson Education Company.

版权所有,翻印必究。

本书封面贴有华中科技大学出版社(原华中理工大学出版社)激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

C++设计新思维/(美)Andrei Alexandrescu 著;侯捷 於春景 译
武汉:华中科技大学出版社,2003年3月
ISBN 7-5609-2906-0

I. C...

Ⅰ. ①A... ②侯... ③於...

Ⅱ. C语言-程序设计

N. TP312

责任编辑:周 筠(<http://yeka.xilubbs.com>)

出版发行:华中科技大学出版社 (武昌喻家山 邮编:430074)

录排:华中科技大学惠友科技文印中心

印刷:湖北新华印务有限公司

开本:787×1092 1/16

印张:21.75 插页:2 字数:400 000

版次:2003年3月第1版

印次:2003年3月第1次印刷

印数:1—8 000

定价:59.80元

ISBN 7-5609-2906-0/TP·501

序言

by Scott Meyers

1991 年，我写下《*Effective C++*》第一版。那本书几乎没有讨论 `template`，因为它刚刚才被加入语言之中，我对它几乎一无所知。为了书中包含的一点点 `template` 代码，我曾通过电子邮件请别人验证，因为我手上的编译器都没有提供对 `template` 的支持。

1995 年，我写下《*More Effective C++*》。又一次，我几乎没有讲述 `template`。这一次阻止我的，既不是对 `template` 知识的缺乏（在那本书的初稿中，我曾打算以一整章讲述 `template`），也不是我的编译器在这方面有所缺陷。真正的理由是我担心，C++ 社群对 `template` 的理解即将经历一次巨大的变化，我对它所说的任何事情，也许很快就会被认为是陈旧的、肤浅的，甚至完全错误的。

我的担心出于两个原因。第一个原因和 John Barton 及 Lee Nackman 在 C++ *Report* 1995 年 1 月的一篇专栏文章有关。这篇文章讨论的是：如何经由 `template` 执行型别安全的维度分析，同时做到运行期零成本。我自己也曾在这个问题上花了不少时间，而且我知道很多人也在寻找解答，但没有人成功。Barton 和 Nackman 的创新解法让我认识到，`template` 在太多的地方有用，不只是用来生成“`T` 容器”。

以下是他们的设计示例。这段代码对两个物理量作乘法运算，而这两个物理量具有任意维数的型别：

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator*(Physical<m1, l1, t1> lhs,
                                         Physical<m2, l2, t2> rhs)
{
    return Physical<m1+m2, l1+l2, t1+t2>::unit*lhs.value()*rhs.value();
}
```

即使我没有说明这段代码，但有一点很清楚：这个 `function template` 有 6 个参数，可没有一个是型别！`template` 的这种用法对我来说是头一次见到，我确实有点目眩。

不久之后，我开始阅读 STL。在 Alexander Stepanov 精巧的程序库设计中，容器（containers）对算法（algorithms）一无所知，算法亦对容器一无所知；迭代器（iterator）的行为像指针（但

却有可能是对象)：容器和算法像接受函数指针一样地接受函数对象 (function object)；用户可以扩充程序库，但不必继承其中任何 base class，也不必重新定义任何 virtual function。这一切都让我觉得——就像当初我看到 Barton 和 Nackman 的成果那样——我对 template 几乎一无所知。

所以，在《*More Effective C++*》中，我几乎没有提到 template。我还能怎样？我对 template 的认识还停留在“τ 容器”阶段，而 Barton、Nackman、Stepanov，还有其他人早已证明，那种用法只不过刚刚触到 template 的皮毛而已。

1998 年，Andrei Alexandrescu 和我开始了电子邮件交流；不久之后我意识到，我得再次修正我对 template 的认识。Barton、Nackman、Stepanov 让我感到震惊的是：template 可以“做什么”；而 Andrei 的成果最初给我的印象是：template “如何”完成它所做的事情。

在 Andrei 协助推广的很多工具中，有这样一个最简单的东西：当我向人们介绍 Andrei 的工作时，我也一直将这当做一个例子。这就是 CTAssert template，作用和 assert 宏类似，但施行于“可在编译期间被核定 (evaluated)”的条件句中。以下便是 CTAssert template：

```
template<bool> struct CTAssert;  
template<> struct CTAssert<true> {};
```

仅此而已。请注意，这个 CTAssert 从来没被定义。请注意，它有一个针对 true (而非 false) 的特化体。在这个设计中，“缺少”的东西至少和提供的东西一样重要。它让你以一种新的方式看待 template，因为大部分“源码”被刻意遗漏了。和我们大多数人以往的想法相比，这是一种极为不同的思维方式。(本书之中 Andrei 讨论了一个更为复杂的 CompileTimeChecker template，而不是 CTAssert)

后来，Andrei 将注意力转移到 idioms (惯用手法) 和 design patterns (设计模式，尤其是 GoF⁴ 模式) 的开发上，提供了 template-based 实作品。这导致他和模式社群的一场短暂冲突，因为后者信奉一条基本原则：模式 (patterns) 无法以代码表述。一旦弄清 Andrei 是在致力于使模式的实现得以自动化，而非试图将模式本身以代码来表述，反对声音也就消失了。我很高兴看到，Andrei 和 GoF 之一 John Vlissides 达成了合作：在 *C++ Report* 上，他们就 Andrei 的研究成果推出了两个专栏。

在开发 templates 用以产生 idioms (惯用手法) 和 design patterns (设计模式) 实作品时，所有实作者需要面对的各种设计抉择，Andrei 也都必须面对。代码应该做到多线程安全吗？辅助存储器应当来自 heap 或是 stack 抑或 static pool？提领 smart pointers 之前是否应该针对 null 进行检查？程序关闭时如果 Singleton 的析构函数试图使用另一个已被摧毁的 Singleton，会发生什么事？Andrei 的目标是：为用户提供所有可能的的设计选择，但不强制任何东西。

⁴ "GoF" 意味着 "Gang of Four"，指的是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，他们四人是模式 (patterns) 权威书籍《*Design Patterns: Elements of Resusable Object-Oriented Software*》(Addison-Wesley, 1995) 的作者。

Andrei 的方案是：将这种选择以 `policy classes` 的形式封装起来，允许客户将 `policy classes` 当做 `template` 参数传递，同时为这种 `classes` 提供合理的缺省值，使大多数客户可以忽略这些参数。其结果令人瞠目结舌。例如本书的 `SmartPointer` `template` 只有 4 个 `policy` 参数，但它可以生成 300 多个不同的 `smart pointer` 型别，每一个都具有不同的行为特征！满足于 `smart pointer` 缺省行为的程序员可以忽略 `policy` 参数，只需指定“`smart pointer` 所指对象”之型别，从而获得精心制作的 `smart pointer class` 所带来的好处。嗯，不费吹灰之力。

最后要说的是，本书叙述了三个不同的技术故事，每一个都引人入胜。首先，它就 C++ `template` 的能力和灵活性提供了新的见解——如果第三章的 `typelists` 没有让你感到振奋，一定是因为你暮气沉沉）。第二，它标示了一个正交维度（`orthogonal dimensions`），告诉我们 `idioms` 和 `patterns` 的实现可以不同。对 `templates` 设计者和 `patterns` 实作者而言，这是十分重要的资讯，但是在大多数讲述 `idioms` 或 `patterns` 的文献中你都找不到这方面的研究。第三，`Loki`（本书介绍的 `template library`）源码可以免费下载，所以你可以研究 Andrei 讨论的 `idioms` 和 `patterns` 所对应的 `template` 实际作品。这些代码可以严格检测你的编译器对 `templates` 的支持程度，此外当你开始自己的 `template` 设计时，它还是无价的起点。当然，直接使用 `Loki` 也是完全可以的（而且完全合法），我知道 Andrei 也愿意你运用他的成果。

就我所知，`template` 的世界还在变化，速度之快就像 1995 年我回避写它的时候一样。从发展的速度来看，我可能永远不会写有关 `template` 的技术书籍。幸运的是一些人比我勇敢，Andrei 就是这样一位先锋。我想你会从此书得到很多收获。我自己就得到了很多。

Scott Meyers
September 2000

前卫的意义

侯捷译序

一般人对 C++ templates 的粗浅印象，大约停留在“容器（containers）”的制作上。稍有研究则会发现，templates 衍生出来的 C++ Generic Programming（泛型编程）技术，在 C++ 标准程序库中已经遍地开花结果。以 STL 为重要骨干的 C++ 标准程序库，将 templates 广泛运用于容器（containers）、算法（algorithms）、仿函数（functors）、配接器（adapters）、分配器（allocators）、迭代器（iterators）上，无处不在，无役不与，乃至原有的 class-based iostream 都被改写为 template-based iostream。

彻底研究过 STL 源码（SGI 版本）的我，原以为从此所有 C++ templates 技法都将不出我的理解与经验。但是《Modern C++ Design》在在打破了我的想法与自信。这本书所谈的 template 技巧，以及据以实作出来的 Loki 程序库，让我瞠目结舌，陷入沉思…与…呃…恍惚☹。

本书分为两大部分。首先（第一篇）是基础技术的讨论，包括 template template parameters（请别怀疑，我并没有多写一个字）、policies-based design、compile-time programming、recursive templates, typelists。每一项技术都让人闻所未闻，见所未见。

第二部分（第二篇）是 Loki 程序库的产品设计与实现，包括 Small-Object Allocation¹, Generalization Functors, Singleton, Smart Pointers, Object Factories, Abstract Factory, Visitor, Multimethods。对设计模式²（design patterns）稍有涉猎的读者马上可以看出，这一部分主题都是知名的模式。换言之，作者 Andrei 尝试以 templates-based, policies-based 手法，运用第一篇完成的基础建设，将上述模式具体实现出来，使任何人能够轻松地在 Loki 程序库的基础上，享受设计模式所带来的优雅架构。

¹ Small-Object Allocation 属于底层服务的“无名英雄”，故而在章节组织上仍被划入第一篇。

² patterns 一词，台湾大陆两地共出现三种译法：(1) 范式，(2) 样式，(3) 模式。我个人最喜欢“范式”，足以说明 patterns 的“典范”意味。因此，繁体版以“范式”称 patterns。为尊重大陆术语习惯，简体版以“模式”称 patterns。本书所有 patterns 都保留英文名称并以特殊字型标示，例如 Object Factories, Visitors...

设计模式 (Design Patterns) 究竟能不能被做成“易拉罐”让人随时随地喝上一口，增强体力？显然模式社群 (patterns community) 中有些人不这么认为——见稍后 Scott Meyers 序文描述。我以为，论断事物不由本质，尽好口舌之辩的人，不足取也。Andrei 所拓展的天地，Loki 所达到的高度，不会因为它叫什么名字而有差异，也不会因为任何人加诸它身上的什么文字包装或批评或解释或讨好，而有不同。它，已经在那儿了。

本书涉足无人履踏之境，不但将 C++ templates 和 generics programming 技术做了史无前例的推进，又与 design patterns 达成巧妙的结合。本书所谈的技术，所完成的实际产品，究竟是狂热激进的象牙塔钻研？抑或高度实用的崭新设计思维？作为一个技术先锋，Loki 的现实价值与未来，唯赖你的判断，和时间的筛选。

然而我一定要多说一句，算是对“唯实用论”的朋友们一些忠告。由来技术的推演，并不只是问一句“它有用吗”或“它现在有用吗”可以论断价值的。牛顿发表万有引力公式，并不知道三百年后人们用来计算轨道、登陆月球。即使在讲述“STL 运用”的课堂上，都还有人觉得太前卫，期盼却焦躁不安，遑论“STL 设计思维和内部实作”这种课，遑论 Loki 这般前卫技术。很多人的焦虑是：我这么学这么做这么写这么用，同侪大概看不懂吧，大概跟不上吧。此固值得关注，但个人的成长千万别被群体的惯性绊住脚步³。我们曾经鄙夷的别人的“无谓”前卫，可能只因我们故步自封，陷自己于一成不变的行为模式；或因为我们只看到自家井口的天空。当然，也可能某些前卫思想和技术，确实超越了庞大笨重迟缓的现实世界的接受度。你有选择。作为一位理性思考者，身在单纯可爱的技术圈内，请不要妄评先锋，因为他实在站在远比你（我）高得太多的山巅上。不当的言语和文字并不能为你（我）堆砌楼台使与同高。

深度 + 广度，古典 + 前卫，理论 + 应用，实验室 + 工厂，才能构筑一个不断进步的世界。

侯捷 2003/01/08 于 台湾·新竹

jjhou@ccca.nctu.edu.tw

<http://www.jjhou.com> (繁)

<http://jjhou.csdn.net> (简)

P.S. 本书译稿由我和於春景先生共同完成。春景负责初译，我负责其余一切。春景技术到位，译笔极好，初译稿便有极佳品质，减轻我的许多负担。循此以往必成为第一流 IT 技术译家。我很高兴和他共同完成这部作品。本书由我定稿，责任在我身上，勘误表由我负责。本书同步发行繁体版和简体版；基于两岸计算机术语的差异，简体版由春景负责必要转换。

P.S. 本书初译稿前三章，邱铭彰先生出力甚多，特此致谢。

P.S. STL, Boost, Loki, ACE... 等程序库的发展，为 C++ 领域掘注了极大活力和竞争力，也使泛型技术在 C++ 领域有极耀眼的发展。这是 C++ 社群近年来最令人兴奋的事。如果你在 C++ 环境下工作，也许这值得你密切关注。

³ 从万有引力观之，微小粒子难逃巨大质量团的吸滞（除非小粒子拥有高能量）。映照人生，这或许是一种悲哀。不过总会有那么一些高能粒子逸脱出来——值得我们转悲为喜，怀抱希望。

译序

by 於春景

三年前，当我第一次接触 `template` 的时候，我认为那只不过是一位“戴上了新帽子”的旧朋友：在熟悉的 `class` 或 `function` 的头顶上，你只需扣上那顶古怪的尖角帽——添上一句 `template <class T1, ...>`——然后将熟悉的数据型别替换为 `T1, T2...`，一个 `template` 就摇身而至！嗯，我得承认，戴上了帽子的 `template` 的确是个出色的代码生成器，好似具有滋生代码“魔法”的 `macro`，但毕竟还不能成其为“戴上了帽子的魔术师”。

后来，我开始学习 GP（generic programming）和 STL（standard template library）。我不禁哑然。在 GP 领域，`template` 竟扮演着如此重要的角色，以至于成为 C++ GP 的基石。在 GP 最重要的商业实作品 STL 中，`template` 向我们展示其无与伦比的功效。回想起自己当初对 `template` 的比喻，哑然失笑之余，我惊叹 `template` 在 GP 和 STL 中将自己的能力发挥到了“极致”。

然而，这一次，《*Modern C++ Design*》又让我默然。我不得不承认，Andrei Alexandrescu 的这部著作（及其 Loki library）带给我的，是对 `template` 和 GP 技术又一次震撼般的认识！

这种震撼感受，源于技术层面，触及设计范畴。`template` 的技术核心在于编译期动态机制。与运行期多态（runtime polymorphism）相比，这种动态机制提供的编译期多态特性，给了程序运行期无可比拟的效率优势。本书中，Andrei 对 `template` 编译期动态机制的运用可谓淋漓尽致。以 `template` 打造而成的 `typelist`、`small-object allocator`、`smart pointer` 不仅具有强大功能，而且体现了无限的扩充性；将 `template` 技术大胆地运用到 `design patterns` 中，更为 `design patterns` 的实现提供了灵活、可复用的泛型组件。

在这些令人目眩的实作技术之后，蕴涵着 Andrei 倡导并使用的 `policy-based` 设计技术。利用这一耳目一新的设计思想，用户代码不再仅仅是技术实作上的细节，你甚至可以让代码在编译期作出设计方案的选择！这种将“设计概念”和“`template` 编译期多态”结合起来的设计思维，将 C++ 程序设计技术提升到了新的高度，足以振聋发聩。

也许只有时间才能证实, Andrei 为我们展示的, 或许是 C++ 程序设计技术的一次革命; 在 C++ 的历史上, 《*Modern C++ Design*》将是一部重要的著作。Anrei 对 `template`、`generic programming` 技术, 以及 `template` 在 `design patterns` 中的运用等课题所做的深入阐释和大胆实践, 可谓前无古人。

遗憾的是, 在当今主流 C++ 编译器上, Loki 很难顺利通过编译。例如面对 "template template parameter" 的“难题”, 很多编译器毫无招架之力。应该说, 这并不是 Andrei 和 Loki 过于超前, 而是 C++ 编译器应当迅速跟进。这意味着作为 C++ 程序员的你我, 也应当迅速跟进!

作为 C++ 程序员的我, 已从此书获益良多。这是一部让我在翻译过程中毫不感到倦怠的巨著。它时时引发我思索, 给我以启迪, 并让我重拾研习 C++ 的快乐。这得感谢 Andrei。在这样一部讲述高级技术的专著中, Andrei 的讲解细致深入, 条理得当, 语言却又极为简明清晰。我期望中文版能保留这一特色。

除了作者之外, 在翻译本书的过程中, 给我更多教益的还有侯捷先生。我的初译稿便是在先生不断的鼓励和指导下完成的。先生谦和的人品和技术上的深邃见解, 令我钦佩和谨记。还要感谢周筠 (yeka) 编辑, 我的每一本译作都离不开您的参与和悉心帮助, 本书也不例外。最后, 感谢所有关心我的朋友, 愿你们也像我一样喜爱这本书。

於春景 2002/12/15

深圳蛇口, 海上世界

billyu@lostmouse.net

序言

by John Vlissides

关于 C++，还有什么没有说到的？唔，很多，本书所谈的一切几乎都是。本书提供的是编程技术——generic programming、template metaprogramming、OO programming、design patterns——的融合。这些技术分开来可以有良好的理解，但对于它们之间的协作关系，我们才刚刚开始认识。这些协同作用为 C++ 打开了全新视野，而且不仅仅在编程方面，还在于软件设计本身；对软件分析和软件体系结构来说，它也具有丰富的内涵。

Andrei 的泛型组件将抽象层次提升到了新的高度，足以使 C++ 在各方面看起来像是一种设计规格（design specification）语言。但是，不同于专用的设计语言，你还保有 C++ 全部的表达性和对它的驾轻就熟。Andrei 向你展示如何根据设计思想——singletons、visitors、proxies、abstract factories... ——来编写程序。甚至你可以经由 template 参数改变实作选择，而且几乎没有运行期开销。你不必求助于新的开发工具，也不必学习晦涩难懂的方法学（methodology）。你需要的只是一个可靠的、新型的 C++ 编译器，以及本书。

多年来，代码生成器（code generators）一直有类似承诺，但我自己的研究以及实践经验使我相信，最终，代码生成器无法匹敌。你会有“往返旅程（round-trip）”问题，“缺乏值得生成的代码”问题，“生成器不灵活”问题，“生出莫名其妙的代码”问题，当然还有“无法将自己的代码和该死的生成出来的代码整合在一起”的问题。这些问题中的任何一个都有可能成为绊脚石；而且，对大多数编程挑战而言，这些绊脚石都使得“代码自动生成”不可能成为一种解决方案。

如果能获得“代码自动生成”理论上的好处——快速、易开发、冗余降低、错误更少——而又没有它们的缺点，该有多好！这正是 Andrei 的做法所承诺的。在易于使用、可相互混合和匹配的 templates 中，泛型组件实现了出色的设计。它们完成的几乎就是代码生成器的功能：产生供编译器使用的规范代码（boilerplate code）。差别在于它们是在 C++ 之内（而非之外）完成这些功能。成果是“与应用代码的无缝整合”。同时你还是可以运用 C++ 语言的全部威力，对设计进行扩充、改写或者调整，从而符合你的需要。

无可否认，这里的一些技术很复杂，因而难以领会，特别是第 3 章的 `template metaprogramming` 部分。但是一旦你掌握了它，你就奠定了泛型组件架构（`generic componentry`）的坚实基础；后续章节中的各个泛型组件几乎就是自己构造自己。事实上我认为第 3 章关于 `template metaprogramming` 的内容就值回本书的价格，何况还有另外 10 个充满见地、让你获益匪浅的章节。"10" 其实是代表一个数量级。而我确信，你获得的回报会比这个数量还多得多。

John Vlissides

IBM T.J. Watson Research

September 2000

前言

Preface

也许你正在书店里捧着这本书，问自己该不该买下它。或者，你正在公司的图书室里，犹豫该不该花时间阅读它。我知道你时间宝贵，所以我开门见山。如果你曾经问过自己：如何撰写更高级的 C++ 程序？如何应付即使在很干净的设计中仍然像雪崩一样发生的不相干细节？如何构建可复用组件，使得每次将这些组件应用到下一个程序时都无需对它们大动干戈？如果你曾这样问过自己，那么，本书正是为你所写。

想象这样的情景。你刚从一次设计会议回来，带着一些打印图表，上面有你潦草写下的注解。哦，对象之间传递的事件型别 (event type) 不再是 `char` 而是 `int` 了，于是你修改一行代码。指向 `widget` 的 `smart pointers` 太慢了，得取消一些检查措施，让它们快一点，于是你又修改一行代码。另一个部门刚才添加 `Gadget class`，你的 `object factory` 必须支持它，于是你再次改动一行代码。

你修改了这个设计。编译，链接，搞定。

且慢，场景有点问题，不是吗？现实情形更可能是：你匆匆从会议中赶回来，因为有一大堆工作要做。于是你开始地毯式搜索，并在代码上大动干戈：添加新的代码、引入臭虫、消除臭虫……这就是程序员的生活，不是吗？本书也许不能保证你实现第一场景，但它朝着那个方向迈出坚实的一步。它对软件设计师展示的 C++，宛如一种新语言。

传统上，代码是软件系统中最琐碎、最复杂的环节。尽管历史上出现了各种层次的编程语言，支持各种设计方法（譬如面向对象方法），但在蓝图和代码之间，总是横亘着一条鸿沟。这是因为，代码必须仔细关照具体实现和某些辅助性任务中极其细节的问题。因此，设计意图往往被无尽的细节吞噬。

本书提供了一组可复用的设计产品——所谓“泛型组件”，以及设计这些组件所需要的技术。这些泛型组件为用户带来的明显好处，集中于程序库方面，而处于更广泛的系统体系结构空间中。本书提供的编程技术和实作品 (implementation) 所反映的任务和议题，传统上属于设计范

畴之中，是编写代码之前必须完成的东西。由于身处较高层次，泛型组件就有可能以一种不同寻常但简洁、易于表达、易于维护的方式，将复杂的体系结构反映到代码中。

这里结合了三个要素：设计模式 (design patterns)、泛型编程 (generic programming)、C++。结合这些要素后，我们获得极高层次的可复用性，无论是横向或纵向。从横向空间来看，少量 library code 就可以实现组合性的、实质上具有无穷数量的结构和行为。从纵向空间来看，由于这些组件的通用性，它们可广泛应用于各种程序中。

本书极大地归功于设计模式 (design patterns)——面临面向对象程序开发中的常见问题时，它是强有力的解决方案。设计模式是经过提炼的出色设计方法，对于很多情况下碰到的问题，它都是合理而可复用的解决方案。设计模式致力于提供深具启发、易于表达和传递的设计词汇。它们所描述的，除了问题 (problem) 之外，还有久经考验的解法及其变化形式，以及选择每一种方案所带来的后果。设计模式超越了任何一种设计语言所能表达的东西——无论那种语言多么高级。本书遵循并结合某些设计模式，提供的组件可以解决广泛的具体问题。

泛型编程是一种典范 (paradigm)，专注于将型别 (type) 抽象化，形成功能需求方面的一个精细集合，并利用这些需求来实现算法。由于算法为其所操作的型别定义了严格、精细的接口，因此，相同的算法可以运用于广泛的型别集 (a wide collection of types)。本书提供的实作品采取泛型编程技术，以最小代价获得足以和手工精心编写的代码相匹敌的专用性、高度简洁和效率。

C++ 是本书使用的唯一工具。在本书中，你不会看到漂亮的窗口系统、复杂的网络程序库或灵巧的日志记录 (logging) 机制。相反的，你会发现很多基础组件；这些组件易于实现以上所有系统（甚至更多）。C++ 具有实现这一切所需要的广度，其底层的 C 内存模型保证了最原始效率 (raw performance)，对多态 (polymorphism) 的支持成就了面向对象技术，templates 则展现为一种令人难以置信的代码生成器。Templates 遍及本书所有代码，因为它们可以令用户和程序库之间保持最密切的协作。在遵循程序库约束的基础上，程序库的用户可以完全控制代码的生成方式。泛型组件库的角色在于，它可以让用户指定的型别和行为，与泛型组件结合起来，形成合理的设计。由于所采技术之静态特性，在结合和匹配相应组件时，产生的错误通常在编译期便得以发现。

本书最明显的意图在于创建泛型组件，这些组件预先实现了设计模块，主要特点是灵活、通用、易用。泛型组件并不构成 framework。实际上它们采用的做法是互补性的；虽然 framework 定义了独立的 classes，用来支持特定的对象模型，但泛型组件(s) 是轻量级设计工具，互相独立，可自由组合和匹配。实现 frameworks 时泛型组件可带来很大帮助。

本书读者

本书预定的读者主要分为两类。第一类是富有经验的 C++ 程序员，他们希望经由本书掌握最先进的程序库编写技术。本书提供了新而强大的 C++ 技术，这些技术具有惊人能力，有一些甚至令人匪夷所思。这些技术对于撰写高级程序库极有帮助。当然，希望更上层楼的中阶 C++ 程序员也会发现本书十分有益，特别是如果他们愿意付出一些毅力。本书虽然有时给出一些高难度的 C++ 代码，但都有详尽说明。

本书预定的第二类读者是繁忙的程序员，他们需要完成工作，但无法投入时间进行深入学习。他们可以快速略过最复杂的细节，把注意力放在如何使用本书提供的程序库上。每一章都有一个导入说明，并以概览 (Quick Facts) 结束。程序员们会发现这种安排方式为理解和使用本书组件提供了有益的参考。这些组件可以分开理解，它们功能强大但很安全，而且让人乐于使用。

你需要扎实的 C++ 经验，以及强烈的求知欲。你也需要对 templates 和 STL (Standard Template Library) 有一定的掌握。

如果你已经了解 design patterns (Gamma 等著, 1995)，那当然好，但并非必要。书中对于用到的 patterns 和 idioms (惯用手法) 都有详细介绍。本书并不是 patterns 方面的专著，并不试图完整阐述 patterns。由于 patterns 是程序库设计者从实践的角度提出的，所以即使那些曾经关注 patterns 的读者也会发现，他们的视野如今有了更新——如果他们曾经受到束缚的话。

Loki

本书讲述一个实际的 C++ 程序库，称为 Loki。Loki 是挪威神话中的智慧之神，同时也是一个淘气鬼；作者希望，这个程序库的创意和灵活会让你想起那个有趣的挪威神话人物。程序库中的所有元素都位于名字空间 (namespace) Loki 之内；这一名称并未出现于书中范例程序上，因为那会为代码带来非必要的缩排格式，并增加代码的数量。Loki 是免费的，你可以从 <http://www.awl.com/cseng/titles/0-201-70431-5> 下载它。

除了线程 (threading) 部分，Loki 完全以标准 C++ 写成。唉，这也意味着目前很多编译器无法处理其中的某些部分。我在 Metrowerks CodeWarrior Pro 6.0 和 Comeau C++ 4.2.38 上实作并测试过 Loki，并且都在 Windows 平台上。KAI C++ 处理 Loki 代码好象也没有问题。随着供应商逐渐发行更新更好的编译器，你将能够运用 Loki 提供的所有功能。

本书提供的 Loki 代码和范例采用了一种很普及的写码标准，这一标准最早由 Herb Sutter 倡导。我相信你很快便能适应它。简单地说：

- classes、functions、枚举型别 (enumerated type) 看起来像 LikeThis。
(译注：由于版面上的需要，中译本的 functions 看起来像 LikeThis)
- 变量和枚举值看起来像 likeThis。
- 成员变量看起来如 likeThis_。
- template 参数如果只可能是用户自定义型别，那么它会被声明为 class；如果还可能是基本型

别，那么它会被声明为 `typename`。

内容组织

本书由两大篇组成：技术篇和组件篇。第一篇（1~4 章）讲述的是，在泛型编程中——特别是在泛型组件的构造中——所运用的 C++ 技术。它展示了与 C++ 相关的大量功能和技术：policy-based 设计、partial template specialization、typelists、local classes 等等。你可以按部就班地阅读本篇，然后回过头来参考特定章节。

第二篇建立在第一篇的基础上，实作出多个泛型组件。这些并非纸上谈兵：他们是具有工业强度的组件，可应用于现实世界的应用程序中。C++ 开发者在日常工作中经常遇到的议题，例如 smart pointers、object factories、functor objects，在此都有深入的探讨，并提供泛型实作。文中提供的实作品满足了基本需要，解决了基本问题。本书并不讲述这一块那一块代码做些什么，它采行的方法是：讨论问题，选择设计决策，然后逐步实现这些设计决策。

第 1 章提供的是 policies，一种有助于产生灵活设计的 C++ 技巧。

第 2 章讨论和泛型编程有关的通用 C++ 技巧。

第 3 章实作 typelists，一种功能强大、用于操纵型别的数据结构。

第 4 章介绍一个重要的辅助工具：小型对象分配器。

第 5 章介绍泛化仿函数的概念；在运用 Command 模式的设计中，它很有用处。

第 6 章讲述 Singleton 对象。

第 7 章讨论和实现了 Smart Pointers。

第 8 章讲述 generic Object Factories。

第 9 章探讨 Abstract Factory 设计模式，并提供一份实作品。

第 10 章以泛型方式实现了 Visitor 设计模式的几个变型。

第 11 章实现了数个 MultiMethod 引擎；这些方案体现设计上的各种选择。

“设计”涵盖许多重要工作，C++ 程序员必须以规律的、标准的、合格的基础和准则来对付。我个人认为 Object Factories（第 8 章）是所有高品质多态设计（polymorphic design）的基石。Smart Pointers（第 7 章）是大大小小许多 C++ 应用程序的重要组成部分。Generalized Functors（第 5 章）有极为广泛的应用，一旦你拥有它，许多复杂的设计问题都能够迎刃而解。其他更特殊的泛型组件，例如 Visitor（第 10 章）或 MultiMethod（第 11 章），也都有重要而合适的应用，并将语言的支持推向极致。

致谢

Acknowledgements

我要感谢我的父母，他们勤劳地度过了那段最为漫长艰辛的岁月。

我要特别强调的是，这本书，连同我的大部分职业生涯，如果没有 Scott Meyers，就都不会存在。自 1998 年于 C++ World Conference 结识 Scott 以来，他就一直帮助我，使我做得更多更好。Scott 第一个热情地鼓励我，让我将我的早期想法付诸实践。他将我引荐给 John Vlissides，促成了另一个具有丰硕成果的合作；他说动 Herb Sutter，让我成为 C++ Report 的专栏作家；他将我介绍给 Addison-Wesley 出版公司，实质上强迫着我开始这本书的写作，而那时我对纽约的销售人员一点都不了解。整本书的创作过程中，Scott 一直帮助我，给我审阅和建议，和我分享写作的痛苦，但没有得到任何好处。

多谢 John Vlissides，他不但提出深邃的见解，让我相信我的方案中存在问题，还为我提出了更好的方案。第 9 章之所以存在，正是因为 John 坚持“事情可以做得更好”。

感谢 P.J. Plauger 和 Marc Briand，他们鼓励我为 C/C++ User Journal 撰写文章，那时我以为专栏作家是外星人。

感谢我的编辑 Debbie Lafferty，她给了我不断的支持，并提出敏锐的建议。

我在 RealNetworks 的同事，特别是 Boris Jerkunica 和 Jim Knaack，给我很大的帮助；他们为我营造了自由、竞争、向上的气氛。我为此感谢他们。

我也十分感激 comp.lang.c++.moderated 和 comp.std.c++ Usenet 新闻群组的所有参与者。这些朋友极大地促进了我对 C++ 的认识。

我还要把我的感谢献给本书初稿审阅者：Mihail Antonescu, Bob Archer (书稿的最完整审阅者), Allen Broadman, Ionut Burete, Mirel Chirita, Steve Clamage, James O. Coplien, Doug Hazen, Kevlin Henney, John Hickin, Howard Hinnant, Sorin Jianu, Zoltan Kormos, James Kuyper, Lisa Lippincott, Jonathan H. Lundquist, Petru Marginenean, Patrick McKillen, Florin Mihaila, Sorin Oprea, John potter,

Adrian Rapiteanu, Monica Rapiteanu, Brian Stanton, Adrian Steflea, Herb Sutter, John Torjo, Florin Trofin, Cristi Vlasceanu。他们投入了大量的精力阅读初稿并提出建议。没有他们，本书的品质将大打折扣。

感谢 Greg Comeau，他免费提供给我第一流的 C++ 编译器。

最后，我非常感谢我的所有家人和朋友，感谢他们无尽的鼓励和支持。

目录

Contents

译序 by 侯捷	i
译序 by 於春景	iii
目录	v
序言 by Scott Meyers	xi
序言 by John Vlissides	xv
前言	xvii
致谢	xxi
第一篇 技术 (Techniques)	1
第 1 章 基于 Policy 的 Class 设计 (Policy-Based Class Design)	3
1.1 软件设计的多样性 (Multiplicity)	3
1.2 全功能型 (Do-It-All) 接口的失败	4
1.3 多重继承 (Multiple Inheritance) 是救世主?	5
1.4 Templates 带来曙光	6
1.5 Policies 和 Policy Classes	7
1.6 更丰富的 Policies	12
1.7 Policy Classes 的析构函数 (Destructors)	12
1.8 通过不完全具现化而获得的选择性机能	13
1.9 结合 Policy Classes	14
1.10 以 Policy Classes 定制结构	16
1.11 Policies 的兼容性	17
1.12 将一个 Class 分解为一堆 Policies	19
1.13 摘要	20

第2章 技术 (Techniques)	23
2.1 编译期 (Compile-Time) Assertions	23
2.2 Partial Template Specialization (模板偏特化)	26
2.3 局部类 (Local Classes)	28
2.4 常整数映射为型别 (Mapping Integral Constants to Types)	29
2.5 型别对型别的映射 (Type-to-Type Mapping)	31
2.6 型别选择 (Type Selection)	33
2.7 编译期间侦测可转换性 (Convertibility) 和继承性 (Inheritance)	34
2.8 type_info 的一个外覆类 (Wrapper)	37
2.9 NullType 和 EmptyType	39
2.10 Type Traits	40
2.11 摘要	46
第3章 Typelists	49
3.1 Typelists 的必要性	49
3.2 定义 Typelists	51
3.3 将 Typelist 的生成线性化 (linearizing)	52
3.4 计算长度	53
3.5 间奏曲	54
3.6 索引式访问 (Indexed Access)	55
3.7 查找 Typelists	56
3.8 附加元素至 Typelists	57
3.9 移除 Typelist 中的某个元素	58
3.10 移除重复元素 (Erasing Duplicates)	59
3.11 取代 Typelist 中的某个元素	60
3.12 为 Typelists 局部更换次序 (Partially Ordering)	61
3.13 运用 Typelists 自动产生 Classes	64
3.14 摘要	74
3.15 Typelist 要点概览	75
第4章 小型对象分配技术 (Small-Object Allocation)	77
4.1 缺省的 Free Store 分配器	78
4.2 内存分配器的工作方式	78
4.3 小型对象分配器 (Small-Object Allocator)	80
4.4 Chunks (大块内存)	81
4.5 大小一致 (Fixed-Size) 的分配器	84
4.6 SmallObjAllocator Class	87
4.7 帽子下的戏法	89

4.8	简单, 复杂, 终究还是简单	92
4.9	使用细节	93
4.10	摘要	94
4.11	小型对象分配器 (Small-Object Allocator) 要点概览	94
第二篇	组件 (Components)	97
第 5 章	泛化仿函数 (Generalized Functors)	99
5.1	Command 设计模式	100
5.2	真实世界中的 Command	102
5.3	C++ 中的可调用体 (Callable Entities)	103
5.4	Functor Class Template 骨干	104
5.5	实现“转发式” (Forwarding) Functor::operator()	108
5.6	处理仿函数	110
5.7	做一个, 送一个	112
5.8	引数和返回型别的转换	114
5.9	处理 Pointer to member function (成员函数指针)	115
5.10	绑定 (Binding)	119
5.11	将请求串接起来 (Chaining Requests)	122
5.12	现实世界中的问题之 1: 转发函数的成本	122
5.13	现实世界中的问题之 2: Heap 分配	124
5.14	通过 Functor 实现 Undo 和 Redo	125
5.15	摘要	126
5.16	Functor 要点概览	126
第 6 章	Singletons (单件) 实作技术	129
6.1	静态数据 + 静态函数 != Singleton	130
6.2	用以支持 Singletons 的一些 C++ 基本手法	131
6.3	实施“Singleton 的唯一性”	132
6.4	摧毁 Singleton	133
6.5	Dead (失效的) Reference 问题	135
6.6	解决 Dead Reference 问题 (I): Phoenix Singleton	137
6.7	解决 Dead Reference 问题 (II): 带寿命的 Singletons	139
6.8	实现“带寿命的 Singletons”	142
6.9	生活在多线程世界	145
6.10	将一切组装起来	148
6.11	使用 SingletonHolder	153
6.12	摘要	155
6.13	SingletonHolder Class Template 要点概览	155

第7章	Smart Pointers (智能指针)	157
7.1	Smart Pointers 基础	157
7.2	交易	158
7.3	Smart Pointers 的存储	160
7.4	Smart Pointer 的成员函数	161
7.5	拥有权 (Ownership) 管理策略	163
7.6	Address-of (取址) 操作符	170
7.7	隐式转换 (Implicit Conversion) 至原始指针型别	171
7.8	相等性 (Equality) 和不等性 (Inequality)	173
7.9	次序比较 (Ordering Comparisons)	178
7.10	检测及错误报告 (Checking and Error Reporting)	181
7.11	Smart Pointers to const 和 const Smart Pointers	182
7.12	Arrays	183
7.13	Smart Pointers 和多线程 (Multithreading)	184
7.14	将一切组装起来	187
7.15	摘要	194
7.16	SmartPtr 要点概览	194
第8章	Object Factories (对象工厂)	197
8.1	为什么需要 Object Factories	198
8.2	Object Factories in C++: Classes 和 Objects	200
8.3	实现一个 Object Factory	201
8.4	型别标识符 (Type Identifiers)	206
8.5	泛化 (Generalization)	207
8.6	细节琐事	210
8.7	Clone Factories (克隆工厂、翻制工厂、复制工厂)	211
8.8	通过其他泛型组件来使用 Object Factories	215
8.9	摘要	216
8.10	Factory Class Template 要点概览	216
8.11	Clone Factory Class Template 要点概览	217
第9章	Abstract Factory (抽象工厂)	219
9.1	Abstract Factory 扮演的体系结构角色 (Architectural role)	219
9.2	一个泛化的 Abstract Factory 接口	223
9.3	实作出 Abstract Factory	226
9.4	一个 Prototype-Based Abstract Factory 实作品	228
9.5	摘要	233
9.6	AbstractFactory 和 ConcreteFactory 要点概览	233

第 10 章 Visitor (访问者、视察者)	235
10.1 Visitor 基本原理	235
10.2 重载 (Overloading) : Catch-All 函数	242
10.3 一份更加精练的实作品: Acyclic Visitor	243
10.4 Visitor 之泛型实作	248
10.5 再论 "Cyclic" Visitor	255
10.6 变化手段	258
10.7 摘要	260
10.8 Visitor 泛型组件要点概览	261
第 11 章 Multimethods	263
11.1 什么是 Multimethods?	264
11.2 何时需要 Multimethods?	264
11.3 Double Switch-on-Type: 暴力法	265
11.4 将暴力法自动化	268
11.5 暴力式 Dispatcher 的对称性	273
11.6 对数型 (Logarithmic) Double Dispatcher	276
11.7 FnDispatcher 和对称性	282
11.8 Double Dispatch (双重分派) 至仿函数 (Functors)	282
11.9 引数的转型: static_cast 或 dynamic_cast?	285
11.10 常数时间的 Multimethods: 原始速度 (Raw Speed)	290
11.11 将 BasicDispatcher 和 BasicFastDispatcher 当做 Policies	293
11.12 展望	294
11.13 摘要	296
11.14 Double Dispatcher 要点概览	297
附录 一个超迷你的多线程程序库 (A Minimalist Multithreading Library)	301
A.1 多线程的反思	302
A.2 Loki 的做法	303
A.3 整数型别上的原子操作 (Atomic Operations)	303
A.4 Mutexes (互斥体)	305
A.5 面向对象编程中的锁定语义 (Locking Semantics)	306
A.6 可有可无的 (Optional) volatile 饰词	308
A.7 Semaphores, Events 和其他好东西	309
A.8 摘要	309
参考书目 (Bibliography)	311
索引 (Index)	313

第一篇

技术

Techniques

1

基于 Policy 的 Class 设计

Policy-Based Class Design

这一章将介绍所谓 policies 和 policy classes, 它们是一种重要的 classes 设计技术, 能够增加程序库的弹性并提高复用性, 这正是 **Loki** 的目标所在。简言之, 具备复杂功能的 policy-based class 由许多小型 classes (称为 policies) 组成。每一个这样的小型 class 都只负责单纯如行为或结构的某一方面 (behavioral or structural aspect)。一如名称所示, 一个 policy 会针对特定主题建立一个接口。在“遵循 policy 接口”的前提下, 你可以采用任何适当的方法来实作 policies。

由于你可以混合并匹配各种 policies, 所以藉由小量核心基础组件 (core elementary components) 的组合, 你可完成一个“行为集” (behaviors set)。

本书许多章节都用到了 policies, 例如第 6 章的泛型类 SingletonHolder 运用 policies 来管理对象寿命和多线程安全 (thread safety)。第 7 章的 SmartPtr 几乎全由 policies 构造出来。第 11 章的双分派引擎 (double-dispatch engine) 运用 policies 决定各种取舍。第 9 章的泛型 Abstract Factory (Gamma et al. 1995) 则运用 policies 来选择不同的生成方法。

本章将说明如何运用 policies 来解决问题, 并提供 policy-based classes design 的详细内容。当你要把 class 分解为 policies 时, 本章也会给你一些忠告。

1.1 软件设计的多样性 (Multiplicity)

软件工程, 也许比其他工程展现出更丰富的多样性。你可以采用多种正确做法完成一件事, 而对与错之间存在无尽的细微差别。每一个新选择都会开启一个新世界。一旦你选择了一个解决方案, 便会有一大堆变化随之涌现, 而且会在每个阶段中持续不停地涌现, 大至系统架构, 小至程序片段。所谓软件设计就是解域空间 (solution space) 中的一道选择题。

让我们考虑一个简单的入门级程序雏型: 一个 Smart Pointer (智能指针, 第 7 章)。这种 class 可被用于单线程或多线程之中, 可以运用不同的 ownership (拥有权) 策略, 可以在安全与速度之间协调, 可以支持或不支持“自动转为内部指针”。以上这些特性都可以被自由组合起来, 而所谓解答, 就是最适合你的应用程序的那个方案。

设计的多样性不断困惑着新手。遭遇一个软件设计问题时, 什么是最好的解法? 是 Events, 还

是 Objects? Observers? Callbacks? Virtuals? Templates? 根据不同的规模和层次, 许多不同的解法似乎一样好。

专业软件设计师与新手的最大不同在于, 前者知道什么可以有效运作, 什么不可以。任何设计结构上的问题, 都有许多合适的解法, 然而它们各有不同规格并且各有优缺点, 对眼前的问题可能适合也可能不适合。白板上可接受的方案, 不一定真有实用价值。

设计一个软件系统很困难, 因为它不断要求你做抉择。而程序设计犹如人生, 抉择是困难的。

好极了, 经验老练的设计师知道怎样的抉择会引领出好的设计。但对新手来说, 每一个设计抉择都开启一扇未知世界之门。有经验的设计者就像一名好棋手, 可以看出好几步棋之后的局势。但这需要时间来学习。或许这就是为何编程 (programming) 才华可能在年轻时就展现出来, 而软件设计 (software design) 才华却常需要更多的时间才能成熟。

除了困惑新手, 设计时“各种决定的组合”也常困扰程序库 (library) 撰写者。为了将有用的设计实作出来, 程序库设计者必须将各种常见情况加以分类融合, 并给出一个开放架构, 如此一来, 应用程序员 (application programmer) 便能根据个别情况组合出他所需要的功能。

更确切地说, 如何在程序库中包装出既富弹性又有优良设计的组件 (components)? 如何让用户自行装配这些组件? 如何在合理大小的代码中对抗“邪恶的”多样性? 这些正是本章乃至本书试图回答的问题。

1.2 全功能型 (Do-It-All) 接口的失败

“在一个全功能型接口下实作所有东西”的做法并不好, 理由很多。

比较重要的负面影响包括智力上的负荷, 体积大小的陡峭爬升, 以及效率考量。庞大的 classes 不能视为成功, 因为它们会导致沉重的学习负荷, 并且有“非必要之大规模”倾向, 使得代码远比手工制作还慢。

一个过于丰富的接口的最大问题, 或许在于缺乏静态型别安全性 (static type safety)。系统架构的一个主要基本原则是: 以“设计”实现某些“原则” (axioms), 例如你不能产生两个 Singleton 对象 (第 6 章) 或产生一个 “disjoint” 族系对象 (第 9 章)。理想上, 一个好的设计应该在编译期强制表现出大部分 constraints (约束条件、规范)。

在一个“无所不包”的大型接口上厉行如此的 constraints 是很困难的。一般来说, 一旦你选择了某一组 design constraints, 大型接口内就只有某些子集能够维持其有效语义。程序库 (library) 的运用向来在“语法有效”和“语义有效”之间存在着缝隙。程序员可能写出愈来愈多的概念, 它们虽然“语法有效”, 却“语义无效”。

举个例子, 考虑以 thread-safety (多线程安全性) 观点来实作 Singleton 对象。如果程序库完全包装了线程, 那么一个特别的、不可移植的多线程系统就无法运用这个 Singleton 程序库。如果这个程序库允许其客户使用未受保护的基本函数 (primitive functions), 那么很可能程序员会因

为写出一些“语法有效”但“语义无效”的程序而破坏了整个设计。

如果程序库将不同的设计实作为各个小型 classes, 每个 class 代表一个特定的罐装解法, 如何? 例如在 smart pointer 例中你会看到 `SingleThreadedSmartPtr`, `MultiThreadedSmartPtr`, `RefCountedSmartPtr`, `RefLinkedSmartPtr` 等等。

这种做法的问题是会产生大量设计组合。例如上述提到的四个 classes 必然导致诸如 `SingleThreadedRefCountedSmartPtr` 这样的组合。如果再加一个设计选项 (例如支持型别转换), 会产生更多潜在组合。这最终会让程序库实作者和使用者受不了。很明显地, 这并不是一个好方法。面对这么多的潜在组合 (近乎指数爬升), 千万别企图使用暴力枚举法。

这样的程序库不只造成大量的智力负荷, 也是极端的严苛而死板。一点点轻微不可测的定制行为 (例如试着以一个特定值为预先构造好的 smart pointers 设初值) 都会导致整个精心制作的 library classes 没有用处。

设计是为了厉行 constraints (约束条件、规范)。因此, 以设计为目标的程序库必须帮助使用者精巧完成设计, 以实现使用者自己的 constraints, 而不是实现预先定义好的 constraints。罐装设计不适用于以设计为目标用途的程序库, 就像魔术数字不适用于一般代码一样。当然啦, 一些“最普遍的、受推荐的”罐装解法将受到大众欢迎——只要客端程序员必要时能够改变它们。

程序库领域中的目前水平令人遗憾: 低阶的通用型程序库和专用型程序库大量存在, 而用来直接辅助设计——这是最高阶结构——的程序库几乎没有。这种情况很矛盾, 因为任何不那么平淡无奇的应用程序都有其自己的设计, 所以一个以设计为目标用途的程序库应该适用于绝大多数应用程序。

Frameworks (框架) 试图填补这样的裂口, 不过这种产品把应用程序限制在特定设计内, 而不是让使用者选择并定制 (customize) 设计。如果程序员需要实作出自己的设计, 他们必须从一开始的 classes, functions...做起。

1.3 多重继承 (Multiple Inheritance) 是救世主?

`TemporarySecretary` classes 继承自 `Secretary` 和 `Temporary`¹, 因此它同时拥有后两者 (秘书和临时雇员) 的特性, 以及其他可能的更多特性。这导致一种想法: 多重继承 (Multiple Inheritance) 可能有助于处理“设计组合”——通过少量的、明确选择后的 based classes。这么一来, 使用者藉由继承 `BaseSmartPtr`, `MultiThreaded` 和 `RefCounted`, 便可制作出具有多线程能力、引用计数功能的智能指针。不过, 任何一位有经验的 class 设计者都知道, 这样天真的设计方式其实无法运作。

¹ 这个例子来自 Bjarne Stroustrup“支持多重继承”的旧论点, 出现于《*The C++ Programming Language*》第一版。从那时起, C++ 就再没有提倡过多重继承。

分析多重继承的失败原因，有助于产生更富弹性的设计，这可以对健全的设计方案提供一些有趣的想法。藉由多重继承机制来组合多项功能，会产生如下问题：

1. 关于技术 (*Mechanics*)。目前并没有一成不变即可套用的代码，可以在某种受控情况下将继承而来的 classes 组合 (assemble) 起来。唯一可组合 `BaseSmartPointer`, `MultiThreaded` 和 `RefCounted` 的工具是语言提供的“多重继承”机制：仅仅只是将被组合的 base classes 结合在一起并建立一组用来访问其成员的简单规则。除非情况极为单纯，否则结果难以让人接受。大多数时候你得小心协调继承而来的 classes 的运转，让它们得到所需的行为。
2. 关于型别信息 (*Type information*)。Based classes 并没有足够的型别信息来继续完成它们的工作。例如，想象一下，你正试着藉由继承一个 `DeepCopy` class 来为你的 smart pointer 实作出深层拷贝 (deep copy)。但 `DeepCopy` 应该具有怎样的接口呢？它必须产生一个对象，而其型别目前未知。
3. 关于状态处理 (*State manipulation*)。base classes 实作之各种行为必须操作相同的 state (译注：意指数据)。这意味着他们必须虚继承一个持有该 state 的 base class。由于总是由 user classes 继承 library classes (而非反向)，这会使设计更加复杂而且变得更没有弹性。

虽然本质上是组合 (combinatorial)，但多重继承无法单独解决设计时的多样性选择。

1.4 Templates 带来曙光

templates 是一种很适合“组合各种行为”的机制，主要因为它们是“依赖使用者提供的型别信息”并且“在编译期才产生”的代码。

和一般的 class 不同，class templates 可以以不同的方式定制。如果想要针对特定情况来设计 class，你可以在你的 class template 中特化其成员函数来因应。举个例子，如果有一个 `SmartPointer<T>`，你可以针对 `SmartPointer<Widget>` 特化其任何成员函数，这可以为你在设计特定行为时提供良好粒度 (granularity)。

犹有进者，对于带有多个参数的 class templates，你可以采用 partial template specialization (偏特化，第2章介绍)。它可以让你根据部分参数来特化一个 class template。例如，下面是一个 template 定义：

```
template <class T, class U> class SmartPtr { ... };
```

你可以令 `SmartPointer<T,U>` 针对 `Widget` 及其他任意型别加以特化，定义如下：

```
template <class U> class SmartPtr<Widget, U> { ... };
```


template 的编译期特性以及“可互相组合”特性，使它在设计期非常引人注目。然而一旦你开始尝试实作这些设计，你会遭遇一些不是那么浅白的问题：

1. 你无法特化结构。单单使用 templates，你无法特化“class 的结构”（我的意思是其数据成员），你只能特化其成员函数。
2. 成员函数的特化并不能“依理扩张”。你可以对“单一 template 参数”的 class template 特化其成员函数，却无法对着“多个 template 参数”的 class template 特化其个别成员函数。例如：

```
template <class T> class Widget
{
    void Fun() { .. generic implementation ... }
};
// OK: specialization of a member function of Widget
template <> void Widget<char>::Fun()    // 译注：原文少了 void
{
    ... specialized implementation ...
}

template <class T, class U> class Gadget
{
    void Fun() { .. generic implementation ... }
};
// Error! Cannot partially specialize a member class of Gadget
// 译注：因为这是 member function 的 Explicit specialization 并无 partial
// specialization 机制。注意这和 class templates 不同！参见 C++ Primer, 16.9 节
template <class U> void Gadget<char, U>::Fun()
{
    ... specialized implementation ...
}
```

3. 程序库撰写者不能够提供多笔缺省值。理想情况下 class template 的作者可以对每个成员函数提供一份缺省实作品，却不能对同一个成员函数提供多份缺省实作品。

现在让我们比较一下多重继承和 templates 之间的缺点。有趣的是两者互补。多重继承欠缺技术（mechanics），templates 有丰富的技术。多重继承缺乏型别信息，而那东西在 templates 里大量存在。Templates 的特化无法扩张（scales），多重继承却很容易扩张。你只能为 template 成员函数写一份缺省版本，但你可以写数量无限的 base classes。

根据以上分析，如果我们将 templates 和多重继承组合起来，将会产生非常具弹性的设备（device），应该很适合用来产生程序库中的“设计元素”（design elements）。

1.5 Policies 和 Policy Classes

Policies 和 Policy Classes 有助于我们设计出安全、有效率且具高度弹性的“设计元素”。所谓 policy，乃用来定义一个 class 或 class template 的接口，该接口由下列项目之一或全部组成：内隐型别定义（inner type definition）、成员函数和成员变量。

Policies 也被其他人用于 **traits** (Alexandrescu 2000a), 不同的是后者比较重视行为而非型别。Policies 也让人联想到设计模式 Strategy (Gamma et al. 1995), 只不过 policies 吃紧于编译期 (所谓 compile-time bound)。

举个例子, 让我们定义一个 policy 用以生成对象: Creator policy 是一个带有型别 T 的 class template, 它必须提供一个名为 Create 的函数给外界使用, 此函数不接受引数, 传回一个 *pointer to* T 。就语义而言, 每当 Create() 被调用就必须传回一个指针, 指向新生的 T 对象。至于对象的精确生成模式 (creation mode), 留给 policy 实作品作为回旋余地。

让我们来定义一个可实作出 Creator policy 的 class。产生对象的可行办法之一就是表达式 new, 另一个办法是以 malloc() 加上 placement new 操作符 (Meyers 1998b)。此外, 还可以采用复制 (cloning) 方式来产生新对象。下面是三种做法的实例呈现:

```
template <class T>
struct OpNewCreator
{
    static T* Create()
    {
        return new T;
    }
};

template <class T>
struct MallocCreator
{
    static T* Create()
    {
        void* buf = std::malloc(sizeof(T));
        if (!buf) return 0;
        return new(buf) T;
    }
};

template <class T>
struct PrototypeCreator
{
    PrototypeCreator(T* pObj = 0)
        : pPrototype_(pObj)
    {}
    T* Create()
    {
        return pPrototype_ ? pPrototype_->Clone() : 0;
    }
    T* GetPrototype() { return pPrototype_; }
    void SetPrototype(T* pObj) { pPrototype_ = pObj; }
private:
    T* pPrototype_;
};
```

任何一个 policy 都可以有无限多份实作品。实作出 policy 者便称为 **policy classes**²，这个东西并不意图被单独使用，它们主要用于继承或被内含于其他 classes。

这里有一个重要观念：policies 接口和一般传统的 classes 接口（纯虚函数集）不同，它比较松散，因为 policies 是语法导向（syntax oriented）而非标记导向（signature oriented）。换句话说，Creator 明确定义的是“怎样的语法构造符合其所规范的 class”，而非“必须实作出哪些函数”。例如 Creator policy 并没有规范 Create() 必须是 static 还是 virtual，它只要求 class 必须定义出 Create()。此外，Creator 也只规定 Create() 应该（但非必须）传回一个指向新对象的指针。因此 Create() 也许会传回 0 或丢出异常——这都极有可能。

面对一个 policy，你可以实作出数个 policy classes。它们全都必须遵守 policy 所定义的接口。稍后你会看到一个例子，使用者选择了一个 policy 并应用到较大结构中。

先前定义三个 policy classes，各有不同的实作方式，甚至连接口也有些不同（例如 PrototypeCreator 多了两个函数 GetPrototype() 和 SetPrototype()）。尽管如此，它们全都定义 Create() 并带有必要的返回型别，所以它们都符合 Creator policy。

现在让我们看看如何设计一个 class 得以利用 Creator policy，它以复合或继承的方式使用先前所定义三个 classes 之一，例如：

```
// Library code
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{ ... };
```

如果 class 采用一个或多个 policies，我们称其为 hosts 或 **host classes**³。上例的 widgetManager 便是“采用了一个 policy”的 host class。Hosts 负责把 policies 提供的结构和行为组成一个更复杂的结构和行为。

当客户端将 widgetManager template 具现化（instantiating）时，必须传进一个他所期望的 policy：

```
// Application code
typedef WidgetManager< OpNewCreator<Widget> > MyWidgetMgr;
```

让我们分析整个来龙去脉。无论何时，当一个 MyWidgetMgr 对象需要产生一个 widget 对象时，它便调用它的 policy 子对象 OpNewCreator<Widget> 所提供的 Create()。选择“生成策略”（Creation policy）是 widgetManager 使用者的权利。藉由这样的设计，可以让 widgetManager 使用者自行装配他所需要的机能。

这便是 **policy-based class** 的设计主旨。

² 这个名称稍许有点不够精确，因为，如你所见，policy 的实作品也可以是 class templates。

³ 虽然 host classes 从技术上来说是 host class templates，但是就让我们固守和注释 2 相同的定义吧。不论 host classes 或 host class templates，都意味着相同的概念。

1.5.1 运用 Template Template 参数实作 Policy Classes

如同先前例子所示, policy 的 template 引数往往是赘余的。使用者每每需要传入 template 引数给 OpNewCreator, 这很笨拙。一般来说, host class 已经知道 policy class 所需的参数, 或是轻易便可推导出来。上述例子中 widgetManager 总是操作 widget 对象, 这种情况下还要求使用者“把 widget 型别传给 OpNewCreator”, 就显得多余而且危险。

这时候程序库可以使用“template template 参数”来描述 policies, 如下所示:

```
// Library code
template <template <class Created> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
    ...
};
// 译注:Created是CreationPolicy的参数, CreationPolicy则是WidgetManager
// 的参数。Widget已经写入上述程序库中, 所以使用时不需要再传一次参数给policy。
```

尽管露了脸, 上述 Created 也并未对 widgetManager 有任何贡献。你不能在 widgetManager 中使用 Created, 它只是 CreationPolicy (而非 widgetManager) 的形式引数 (formal argument), 因此可以省略。

应用端现在只需在使用 widgetManager 时提供 template 名称即可:

```
// Application code
typedef WidgetManager<OpNewCreator> MyWidgetMgr;
```

搭配 policy class 使用“template template 参数”, 并不单纯只为了方便。有时候这种用法不可或缺, 以便 host class 可藉由 templates 产生不同型别的对象。举个例子, 假设 widgetManager 想要以相同的生成策略产生一个 Gadget 对象, 代码如下:

```
// Library code
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
    ...
    void DoSomething()
    {
        Gadget* pW = CreationPolicy<Gadget>().Create();
        ...
    }
};
```

使用 policies 是否会为我们带来一些优势呢? 乍看之下并不太多。首先, Creator policy 的实作本来就十分简短。当然, widgetManager 的作者应该会把“生成对象”的那份代码写成 inline 函数, 并避开“将 widgetManager 建立为一个 template”时可能遭遇的问题。

然而, `policy` 的确能够带给 `WidgetManager` 非常大的弹性。第一, 当你准备具现化 (instantiating) `WidgetManager` 时, 你可以从外部变更 `policies`, 就和改变 `template` 引数一样简单。第二, 你可以根据程序的特殊需求, 提供自己的 `policies`。你可以采用 `new` 或 `malloc` 或 `prototypes` 或一个专用于你自己系统上的罕见内存分配器。`WidgetManager` 就像一个小型的“代码生成引擎” (*code generation engine*), 你可以自行设定代码的产生方式。

为了让应用程序开发人员的日子更轻松, `WidgetManager` 的作者应该定义一些常用的 `policies`, 并且以“`template` 缺省引数”的形式提供最常用的 `policy`:

```
template <template <class> class CreationPolicy = OpNewCreator>
class WidgetManager ...
```

注意, `policies` 和虚函数有很大不同。虽然虚函数也提供类似效果: `class` 作者以基本的 (primitive) 虚函数来建立高端功能, 并允许使用者改写这些基本虚函数的行为。然而如前所示, `policies` 因为有丰富的型别信息及静态连接等特性, 所以是建立“设计元素”时的本质性东西。不正是“设计”指定了“执行前型别如何互相作用、你能够做什么、不能够做什么”的完整规则吗? `Policies` 可以让你在型别安全 (typesafe) 的前提下藉由组合各个简单的需求来产出你的设计。此外, 由于编译器才将 `host class` 和其 `policies` 结合在一起, 所以和手工打造的程序比较起来更加牢固并且更有效率。

当然, 也由于 `policies` 的特质, 它们不适用于动态连结和二进位接口, 所以本质上 `policies` 和传统接口并不互相竞争。

1.5.2 运用 Template 成员函数实作 Policy Classes

另外一种使用“`template template` 参数”的情况是把 `template` 成员函数用来连接所需的简单类。也就是说, 将 `policy` 实作为一般 `class` (“一般”是相对于 `class template` 而言), 但有一个或数个 `templated members`。

例如, 我们可以重新定义先前的 `Creator policy` 成为一个 `non-template class`, 其内提供一个名为 `Create<T>` 的 `template` 函数。如此一来, `policy class` 看起来像下面这个样子:

```
struct OpNewCreator
{
    template <class T>
    static T* Create()
    {
        return new T;
    }
};
```

这种方式所定义并实作出来的 `policy`, 对于旧式编译器有较佳兼容性。但从另一方面来说, 这样的 `policy` 难以讨论、定义、实作和运用。

1.6 更丰富的 Policies

Creator policy 只指定了一个 `Create()` 成员函数。然而 `PrototypeCreator` 却多定义了两个函数，分别为 `GetProtoType()` 和 `SetProtoType()`。让我们来分析一下。

由于 `WidgetManager` 继承了 `policy class`，而且 `GetPrototype()` 和 `SetPrototype()` 是 `PrototypeCreator` 的 `public` 成员，所以这两个函数便被加至 `WidgetManager`，并且可以直接被使用者取用。

然而 `WidgetManager` 只要求 `Create()`；那是 `WidgetManager` 一切所需，也是用来保证它自己机能的唯一要求。不过使用者可以开发出更丰富的接口。

prototype-based Creator policy class 的使用者可以写出下列代码：

```
typedef WidgetManager<PrototypeCreator>
    MyWidgetManager;
...
Widget* pPrototype = ...;
MyWidgetManager mgr;
mgr.SetPrototype(pPrototype);
... use mgr ...
```

如果此后使用者决定采用一个不支持 `prototypes` 的生成策略，那么编译器会指出问题：`prototype` 专属接口已经被用上了。这正是我们希望获得的坚固设计。

如此的结果是很受欢迎的。使用者如果需要扩充 `policies`，可以在不影响 `host class` 原本功能的前提下，从更丰富的功能中得到好处。别忘了，决定“哪个 `policy` 被使用”的是使用者而非程序库自身。和一般多重接口不同的是，`policies` 给予使用者一种能力，在型别安全 (`typesafe`) 的前提下扩增 `host class` 的功能。

1.7 Policy Classes 的析构函数 (Destructors)

有一个关于建立 `policy classes` 的重要细节。大部分情况下 `host class` 会以“`public` 继承”方式从某些 `policies` 派生而来。因此，使用者可以将一个 `host class` 自动转为一个 `policy class`（译注：向上转型），并于稍后 `delete` 该指针。除非 `policy class` 定义了一个虚析构函数（`virtual destructor`），否则 `delete` 一个指向 `policy class` 的指针，会产生不可预期的结果⁴，如下所示：

```
typedef WidgetManager<PrototypeCreator>
    MyWidgetManager;
...
MyWidgetManager wm;
PrototypeCreator<Widget>* pCreator = &wm; // dubious, but legal
delete pCreator; // compiles fine, but has undefined behavior
```

然而如果为 `policy` 定义了一个虚析构函数，会妨碍 `policy` 的静态连结特性，也会影响执行效率。

⁴ 你可以在本书第 4 章“小型对象分配技术” (Small-Object Allocation) 中找到一份精细的讨论。

许多 **policies** 并无任何数据成员，纯粹只规范行为。第一个虚函数被加入后会为对象大小带来额外开销（译注：因为引入一份 **vptr**），所以虚析构函数应该尽可能避免。

一个解法是，当 **host class** 自 **policy class** 派生时，采用 **protected** 继承或 **private** 继承。然而这样会失去丰富的 **policies** 特性（1.6 节）。**policies** 应该采用一个轻便而有效率的解法——定义一个 **non-virtual protected** 析构函数：

```
struct OpNewCreator
{
    template <class T>
    static T* Create()
    {
        return new T;
    }
protected:
    ~OpNewCreator() {}
};
```

由于析构函数属于 **protected** 层级，所以只有派生而得的 **classes** 才可以摧毁这个 **policy** 对象。这样一来外界就不可能 **delete** 一个指向 **policy class** 的指针。而由于析构函数并非虚函数，所以不会有大小或速度上的额外开销。

1.8 通过不完全具现化而获得的选择性机能

事情还可以变得更好。C++ 的一些有趣特性造就了 **policies** 的威力。如果 **class template** 有一个成员函数未曾被用到，它就不会被编译器具体实现出来。编译器不理睬它，甚至也许不会为它进行语法检验⁵。

如此便导致 **host class** 有机会指明并使用 **policy class** 的可选特性。举个例子，让我们为 **WidgetManager** 定义一个 **SwitchPrototype()**：

```
// Library code
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
    ...
    void SwitchPrototype(Widget* pNewPrototype)
    {
        CreationPolicy<Widget>& myPolicy = *this;
        delete myPolicy.GetPrototype();
        myPolicy.SetPrototype(pNewPrototype);
    }
};
```

下面是一些非常有趣的结果：

⁵ 据 C++ *Standard* 所载，“未被使用之 **template functions**”的语法分析层级，由编译器自行决定。至于语义检验，编译器根本不进行。

- 如果你采用一个“支持 prototype”的 Creator policy 来具现化 widgetManager，你便可以使用 SwitchPrototype()。
- 如果你采用一个“不支持 prototype”的 Creator policy 来具现化 widgetManager，并尝试使用 SwitchPrototype()，会出现编译错误。
- 如果你采用一个“不支持 prototype”的 Creator policy 来具现化 widgetManager，并且从未试图使用 SwitchPrototype()，程序是合法的。

这些都意味着 widgetManager 除了可从弹性且丰富的接口得到好处之外，也仍然可以搭配较简单的接口而正常运作——只要你不试着去使用其中某些成员函数。

widgetManager 的作者现在可以这样定义 Creator policy:

Creator 设定一个型别为 T 的 class template，其中列有 Create()，该函数应该传回一个指针指向新生之 T 对象。Creator 实作码可以选择性地定义两个额外函数： T^* GetPrototype() 和 SetPrototype(T^*)，让使用者可以在生成对象时拥有“取得”或“设定”某个 prototype 的机会。这种情况下，widgetManager 于是可能出现一个 SwitchPrototype(T^* pNewPrototype) 函数，可删除目前的 prototype 并设定一个新的 prototype。

与各个 policy class 结合时，这些不完整具现化 (incomplement instantiation) 带给你 (使用者) 有如程序库设计者一般的非凡自由度。你可以实作出“精瘦并带依赖性的” (lean) host classes，它能够使用额外的特性，并能够姿态优雅地将层次降低至纪律性高的最小化 policies。

1.9 结合 Policy Classes

当你将 policies 组合起来时，便是它们最有力的时候。一般而言，一个高度可组装化的 class 会运用数个 policies 来达成其运作上的各个方面。一个程序库使用者可以藉由组合不同的 policy classes 来选择他所需的高阶行为。

举个例子，假设我们正打算设计一个泛型的 smart pointer (第 7 章有完整实作)。假设你分析出两个应被建立为 policies 的设计：threading model (多线程模型) 和 check before dereference (提领前先检验)。于是你实作出一个带有两个 policies 的 class template，名为 SmartPtr:

```
template <
    class T,
    template <class> class CheckingPolicy,
    template <class> class ThreadingModel
>
class SmartPtr;
```

SmartPtr 有三个 template 参数：一个代表 pointee type (被指对象的型别)，另两个是 policies。在 SmartPtr 之中你可以运用两个 policies 组织出一份稳固的实作品。SmartPtr 成为“集成数个 policies”的协调层，而非一成不变的罐装实作品。以这种方式来设计 SmartPtr，便是赋予使用者“以简单的 typedef 对 SmartPtr 进行配置 (configure)”的能力：

```
typedef SmartPtr<Widget, NoChecking, SingleThreaded>
WidgetPtr;
```


在同一个应用程序中，你可以定义并使用数种不同的 smart pointer classes:

```
typedef SmartPtr<Widget, EnforceNotNull, SingleThreaded>
    SafeWidgetPtr;
```

两个 policies 定义如下:

Checking: 这个名为 `CheckingPolicy<T>` 的 class template 必须公开一个 `Check()` 成员函数，可接受一个型别为 `T*` 的左值。当 smart pointer 即将被提领 (dereference) 时会调用 `Check()`，传入被指对象 (pointee object) 并做检查。

ThreadingModel: 这个名为 `ThreadingModel<T>` 的 class template 必须提供一个名为 `Lock` 的内部型别，该型别的构造函数接受一个 `T&` 参数。对一个 `Lock` 对象来说，其生命中所有对此 `T` 对象的操作行为都是串行的 (serialized)。

举个例子，下面是两个 policy classes `NoChecking` 和 `EnforceNotNull` 的实作内容:

```
template <class T> struct NoChecking
{
    static void Check(T*) {}
};
template <class T> struct EnforceNotNull
{
    class NullPointerException : public std::exception { ... };
    static void Check(T* ptr)
    {
        if (!ptr) throw NullPointerException();
    }
};
```

藉由抽换不同的 Checking policy class，你可以实作出各种不同行为。你甚至可以利用缺省值来初始化被指对象 (pointee object) —— 只要传入一个 reference-to-pointer 即可，像这样:

```
template <class T> struct EnsureNotNull
{
    static void Check(T*& ptr)
    {
        if (!ptr) ptr = GetDefaultValue();
    }
};
```

`SmartPtr` 这样使用 Checking policy:

```
template
<
    class T,
    template <class> class CheckingPolicy,
    template <class> class ThreadingModel
>
class SmartPtr
    : public CheckingPolicy<T>
    , public ThreadingModel<SmartPtr>
```

```

{
    ...
    T* operator->()
    {
        typename ThreadingModel<SmartPtr>::Lock guard(*this);
        CheckingPolicy<T>::Check(pointee_);
        return pointee_;
    }
private:
    T* pointee_;
};

```

注意上述同一个函数中对 `CheckingPolicy` 和 `ThreadingModel` 两个 policy classes 的运用。根据不同的 template 引数, `SmartPtr::operator->` 会表现出两种不同的正交 (orthogonal) 行为。这正是 policies 的组合威力所在。

一旦你设法把一个 class 分解成正交的 policies, 便可利用少量代码涵盖大多数行为。

1.10 以 Policy Classes 定制结构

如同 1.4 节所说, templates 的限制之一是, 你无法定制 (customize) class 的结构, 只能定制其行为。然而 policy-based design 支持结构方面的定制。

假设你想支持“非指针形式”的 `SmartPtr`, 例如某些平台上的某些指针也许会以 `handle` 形式呈现, 这是一种用来传给系统函数的整数值, 藉以取得实际指针。为了解决这种情况, 你可以通过一个所谓的 `Structure policy` 将指针的访问“间接化”。`Structure policy` 将指针的存储概念抽象化, 因此它应该提供一个 `PointerType` 型别 (用以代表指针所指对象的型别)、一个 `ReferenceType` 型别 (用以代表指针所指对象的 `reference` 型别), 以及 `GetPointer()` 和 `SetPointer()` 两函数。

不把 `pointer` 型别硬性规定为 `T*`, 这种做法带来重大好处。例如你可以将 `SmartPtr` 应用于非标准指针型别 (例如 `segment` 架构上的 `near` 指针和 `far` 指针), 或者你可以轻松实作出灵巧解法, 诸如 `before` 和 `after` 函数 (Stroustrup 2000a), 这些可能性都非常有趣。

`smart pointer` 的缺省存储形式是一个带有 `Structure policy` 接口的一般指针, 像下面这样:

```

template <class T>
class DefaultSmartPtrStorage // 译注:Loki 无此 class, 但有一个 DefaultSPStorage
{
public:
    typedef T* PointerType;
    typedef T& ReferenceType;
protected:
    PointerType GetPointer() { return ptr_; }
    void SetPointer(PointerType ptr) { ptr_ = ptr; }
};

```

```
private:
    PointerType pointee_;
};
```

实际指针的存储形式已被完全隐蔽于 `Structure` 接口之内。现在, `SmartPtr` 可以运用一个 `Storage` policy 来取代对 `T*` 的聚合 (aggregating) :

```
template
<
    class T,
    template <class> class CheckingPolicy,
    template <class> class ThreadingModel,
    template <class> class Storage = DefaultSmartPtrStorage
>
class SmartPtr;
```

当然, 为了内嵌所需的结构, `SmartPtr` 必须继承自 `Storage<T>` 或聚合 (aggregate) 一个 `Storage<T>` 对象 (译注: 详见第 7 章)。

1.11 Policies 的兼容性

假设你要产生两个 `SmartPtr`: `FastWidgetPtr` 是一个不需检验的指针, `SafewidgetPtr` 则必须在提领 (dereference) 之前先检验。这时有个有趣的问题: 你能将一个 `FastWidgetPtr` 对象指派 (赋值) 给一个 `SafewidgetPtr` 对象吗? 你应该有能力以其他方法指派它们吗? 如果你想实现出这样的功能, 该如何实作?

让我们从推理跨出第一步。`SafewidgetPtr` 比 `FastWidgetPtr` 有更多限制, 因此我们很容易接受“把 `FastWidgetPtr` 转为 `SafewidgetPtr`”的想法。这是因为 C++ 原就支持隐式转换 (implicit conversion), 不过也存在一些限制, 例如 `non-const` 型别转为 `const` 型别。

从另一方面说, “自由地将 `SafewidgetPtr` 对象转换为 `FastWidgetPtr` 对象”是危险的。因为应用程序大都使用 `SafewidgetPtr`, 只有小型且需要考虑速度的核心代码才会考虑使用 `FastWidgetPtr`。只在明确受控的情况下才允许将 `SafewidgetPtr` 转换为 `FastWidgetPtr`, 此举将有助于保持 `FastWidgetPtr` 的最小用量。

Policies 之间彼此转换的各种方法中, 最好又最具扩充性的实作法是以 policy 来控制 `SmartPtr` 对象的拷贝和初始化, 如下所示 (让我们将先前程序简化为只有一个 policy: `Checking`) :

```
template
<
    class T,
    template <class> class CheckingPolicy
>
class SmartPtr : public CheckingPolicy<T>
{
    ...
    template
```

```

<
    class T1,
    template <class> class CP1,
>
SmartPtr(const SmartPtr<T1, CP1>& other)
    : pointee_(other.pointee_), CheckingPolicy<T>(other)
{ ... }
};

```

SmartPtr 实作出一个“接受任何一种 SmartPtr 对象”的 template copy 构造函数。其中粗体字那一行系根据其引数 SmartPtr<T1, CP1> 的内容，将 SmartPtr 的内容初始化。

下面介绍其运作方式(请接续上述构造函数)。假设你有个 ExtendWidget class, 派生自 widget。当你以一个 SmartPtr<ExtendedWidget, NoChecking> 初始化一个 SmartPtr<Widget, NoChecking> 时，编译器会试着以一个 ExtendWidget* 初始化 widget* (这会成功)，然后以一个 SmartPtr<Widget, NoChecking> 初始化 NoChecking。这看起来很可疑，但是别忘了 SmartPtr 派生自其 policy，所以编译器可以轻易知道你想要以一个 NoChecking 初始化一个 NoChecking。整个初始化过程可以良好进行。

接下来就有意思了。假设你打算以一个 SmartPtr<ExtendedWidget, NoChecking> 初始化一个 SmartPtr<Widget, EnforceNotNull>。此时 ExtendedWidget* 被转为 widget*，一如前面所说。然后编译器试图将 SmartPtr<ExtendedWidget, NoChecking> 拿来匹配 EnforceNotNull 构造函数。

如果 EnforceNotNull 实作出可接受 NoChecking 对象的构造函数，那么编译器会找到那个构造函数，完成转换。如果 NoChecking 实作出可将自己转换为 EnforceNotNull 的转型操作符，那么转换也可以进行。除此之外，都会产生编译错误。

如你所见，当你进行 policies 转换时，两边都有弹性。左边你可以实作转型构造函数，右边你可以实作转型操作符。

assignment 操作符也有同样的难缠问题，幸运的是 Sutter 2000 (译注: *Exceptional C++*, 条款 41) 阐述了一种非常漂亮的技术，可以让你根据 copy 构造函数实作出 assignment 操作符。这是一个非常漂亮的手法，你应该读读那篇文章。Loki 的 SmartPtr 也运用了这项技术。

虽然 NoChecking 转换为 EnforceNotNull 或反向转换感觉都十分合理，但有些转换却是一点也不合理。想象将一个 reference-counted 指针转换为一个支持其他“ownership (拥有权) 策略”的指针，将是一场毁灭性的 copy (有点像 std::auto_ptr)。这样的转换造成语义上的错误。所谓 reference counting 是“所有指向同一对象的指针都为大家所知，并且可根据一个独一无二的计数器加以追踪”，一旦你尝试将某个指针设为另一种 ownership policy，你便是破坏了 reference counting 赖以有效运作的不变性 (恒长性)。

总之，ownership 的转换不该是隐式转换，应该特别小心处理。你最好明确调用某个函数来改变“reference-counted 指针”的 ownership policy。唯有源端指针的 reference count 数值为 1，这个函数才有可能转换成功。

1.12 将一个 Class 分解为一堆 Policies

建立 policy-based class design 的最困难部分，便是如何将 class 正确分解为 policies。一个准则就是，将参与 class 行为的设计鉴别出来并命名之。任何事情只要能以一种以上的方法解决，都应该被分析出来，并从 class 中移出来成为 policy。别忘了，湮没于 class 设计之中的 constraints（约束条件）就像湮没于代码中的魔术常数一样不好。

举个例子，让我们考虑 widgetManager class。如果 widgetManager 内部生成新的 widget 对象，生成方式应该推迟至 policy 才确定。如果 widgetManager 打算存储一大群 widget 对象，比较合理的设计是将集合设计为一个 storage policy，除非你对特定的存储机制有强烈偏好。

极端情形下，host class 几乎就是 policies 的集合，它将设计期的全部决定和约束条件都委派（delegates）给 policies，这样的 host class 只不过是涵盖 policies 集合的外层而已，只能处理 policies 组合出来的行为。

过于泛化的 host class 会产生缺点，它会有过多的 template 参数。实用上，4~6 个以上的 template 参数会造成合作上的笨拙。不过如果 host class 打算提供复杂而有用的功能，该有的 template 参数还是要有。

型别定义（也就是 typedef）是运用 policy-based classes 时的一个重要工具。这不仅是为了方便，也可以确保有条理地运用和易维护性。例如下面这个型别定义：

```
typedef SmartPtr
<
    Widget,
    RefCounted,
    NoChecked
>
WidgetPtr;
```

在代码中使用冗长定义的 SmartPtr 而不使用上述简洁的 widgetPtr，实在乏味而令人厌烦。不过，“乏味”与程序的“可维护性”和“可读性”相比，还是小问题。随着设计的演进，widgetPtr 的定义也许会跟着改变，例如也许会改用与“调试期所用之 NoChecking”不同的一个 checking policy。所有程序都采用 widgetPtr 而不采用“硬以 SmartPtr 写出的实作品”是很重要的。其间差别就像“函数”和“功能相等的 inline 函数”一样。虽然技术上 inline 函数做相同的事情，但我们无法在它背后建立抽象性。

当你将 class 分解为 policies 时，找到正交分解（orthogonal decomposition）很重要。正交分解会产生一些彼此完全独立的 policies。你很容易发现一个非正交分解——如果各式各样的 policies 需要知道彼此，那就是了。

举个例子。试想 smart pointer 里的一个 Array policy。它非常简单——规定 smart pointer 是否指向 array。在这个 policy 的定义中，有一个 T& ElementAt(T* ptr, unsigned int index) 成员函数，以及一个类似的 const T 版本。至于 non-array policy，由于并没有定义 ElementAt()，所以如果

有人试图使用它，会出现编译错误。以 1.6 节的话来说，`ElementAt()` 是一个可选用的、丰富接口下的行为。

下面是两个实作出 Array policy 的 policy classes:

```
template <class T>
struct IsArray
{
    T& ElementAt(T* ptr, unsigned int index)
    {
        return ptr[index];
    }
    const T& ElementAt(T* ptr, unsigned int index) const
    {
        return ptr[index];
    }
};
template <class T> struct IsNotArray {};
```

问题是无论 smart pointer 是否指向 array，都会与另一个 policy: destruction (析构) 产生不良互动。是的，你必须使用 `delete` 来摧毁指针所指对象，却必须使用 `delete[]` 来摧毁指针所指的 object array。

两个 policies 之间如果没有互相影响，才称为正交 (orthogonal)。根据这个定义，Array 和 Destroy policies 不是正交。

如果你仍然需要将 array 的生成和摧毁设为独立的 policy，你得建立一个让它们沟通的办法。你必须让 Array policy 除了提供一个函数外，还提供一个 `bool` 常数，并将它传入 Destroy policy。这会使 Array 和 Destroy 的设计变得更复杂，而且不由得多了些约束条件 (constraints)。

非正交的 policies 是不完美的设计，应该尽量避免，因为这样的设计会降低编译期类型安全性 (type safety)，并导致 host class 和 policy class 的设计更加复杂。

如果你必须使用非正交的 policies，请尽可能借着“把 policy class 当做引数传给其他 policy class template function”来降低相依性。这样一来你还是可以从 template-based 接口带来的弹性中获得利益。剩下的缺点就是，policy 必须暴露它的某些实作细节给其他 policy，这会降低封装性。

1.13 摘要

“设计”就是一种“选择”。大多数时候我们的困难并不在于找不到解决方案，而是有太多解决方案。你必须知道哪一组方案可以圆满解决问题。大至架构层面，小至代码片段，都需要抉择。此外，抉择是可以组合的，这给设计带来了可怕的多样性。

为了在合理大小的代码中因应设计的多样性，我们应该发展出一个以设计为导向 (design oriented) 的程序库，并在其中运用一些特别技术。这些被特意构想出来用以支持巨大弹性的代

码产生器，由小量基本设备（primitive device）组合而成。程序库本身供应有一定数量的基本设备。此外，程序库也供应一些用以建立基本设备的规格，因此客端（client）可以建造出自己想要的设备。这基本上使得 policy-based design 成为开放式架构。这些基本设备我们称为 policies，其实作品则被称为 policy classes。

Policies 机制由 templates 和多重继承组成。一个 class 如果使用了 policies，我们称其为 host class，那是一个拥有多个 template 参数（通常是“template template 参数”）的 class template，每一个参数代表一个 policy。Host class 的所有机能都来自 policies，运作起来就像是一个聚合了数个 policies 的容器。

环绕着 policies 而设计出来的 classes，支持“可扩充的行为”和“优雅的机能削减”。由于采用“public 继承”之故，policy 得以通过 host class 提供追加机能。而 host classes 也能运用“policy 提供的选择性机能”实作出更丰富的功能。如果某个选择性机能不存在，host class 还是可以成功编译，前提是该选择性机能未被真正用上。

Policies 的最大威力来自于它们可以互相混合搭配。一个 policy-based class 可以组合“policies 实作出来的某些简单行为”而提供非常多的行为。这极有效地使 policies 成为对付“设计期多样性”的好武器。

通过 policy classes，你不但可以定制行为，也可以定制结构。这个重要的性质使得 policy-based design 超越了简单的型别泛化（type genericity）——后者对于容器类（container classes）效力卓著。

型别转换对 policy-based classes 而言也是一种弹性的表现。如果你采用 policy-by-policy 拷贝方式，每个 policy 都能藉由提供适当的转型构造函数或转型操作符（甚至两者都提供）来控制它自己接受哪个 policies，或它自己可以转换为哪个 policy。

欲将 class 分解为 policies 时，你应该遵守两条重要准则。第一，把你的 class 内的“设计决定”局部化、命名、分离出来。这也许是一种取舍，也许需要以其他方式明智地完成。第二，找出正交的 policies——也就是彼此之间无交互作用、可独立更动的 policies。

2

技术

Techniques

本章呈现许多贯穿本书的 C++ 技术。为了在各式各样的情境 (context) 中都有用, 它们倾向于泛化 (一般化, general) 和可复用 (reusable), 如此便可在其他情境中找出它们的应用。有些技术如 `partial template specialization` (模板偏特化) 是语言本身的特性, 有些如“编译期 assertions”则需藉由代码实作出来。

本章之中你将了解下列这些技术和工具:

- `Partial template specialization` (模板偏特化)
- `Local classes` (局部类)
- 型别和数值之间的映射 (`Int2Type` 和 `Type2Type` class templates)
- 在编译期察觉可转换性 (convertibility) 和继承性
- 型别信息, 以及一个容易上手的 `std::type_info` 外覆类 (wrapper)
- `Select class template`. 这是一个工具, 可在编译期间根据某个 `bool` 状态选择某个型别
- `Traits`, 一堆 traits 技术集合, 可施行于任何 C++ 型别身上

如果分开来看, 每个技术和其所用之代码也许都不怎么样; 它们都由 5~10 行浅显易懂的代码组成。然而这些技术有一个重要特性: 它们没有极限。也就是说, 你可以把它们组合成一个高阶惯用手法 (idioms)。当它们合作时, 便形成一个强壮的服务基础, 可协助我们建立比较强壮的结构。

这些技术都带有范例, 所以讨论起来并不枯燥。阅读本书其余部分时, 也许你会回头参考本章。

2.1 编译期 (Compile-Time) Assertions

随着泛型编程在 C++ 大行其道, 更好的静态检验 (static checking) 以及更好的可定制型错误消息 (customizable error messages) 的需求浮现了出来。

举个例子, 假设你发展出一个用来作安全转型 (safe casting) 的函数。你想将某个型别转为其他型别, 而为了确保原始信息被保留, 较大型别不能转型为较小型别。

```
template <class To, class From>
```

```

To safe_reinterpret_cast(From from)
{
    assert(sizeof(From) <= sizeof(To));
    return reinterpret_cast<To>(from);
}

```

你可以像运用“C++ 内建之型别转换操作”一样地调用上述函数：

```

int i = ...;
char* p = safe_reinterpret_cast<char*>(i);

```

你必须明白指定 `to` 这个 **template** 引数；编译器会根据 `i` 的型别推导出另一个 **template** 引数 `From`。藉由上述的“大小比较” **assertion** 动作，便可确定“目标型别”足以容纳“源端型别”的所有 **bits**。如此一来，上述代码便可达到正确的型别转换⁶，或导致一个执行期 **assertion**。

很显然，我们都希望错误能够在编译期便被侦测出来。一则因为转型动作可能是你的程序中偶尔执行的分支，当你将程序移植到另一个编译器或平台上时，你可能不会记住每一个潜在的不可移植部分（译注：上例 `reinterpret_cast` 就是不可移植的），于是留下潜伏臭虫，而它可能在用户面前让你出丑。

这里有一道曙光。表达式（**expression**）在编译期评估所得结果是个定值（常数），这意味着你可以利用编译器（而非代码）来作检查。这个想法是传给编译器一个语言构造（**language construct**），如果是非零表达式便合法，零表达式则非法。于是当你传入一个表达式而其值为零时，编译器会发出一个编译期错误的信息。

最简单的方式称为 **compile-time assertions**（Van Horn 1997），在 C 和 C++ 语言中都可以良好运作。它依赖一个事实：大小为零的 **array** 是非法的。

```
#define STATIC_CHECK(expr) { char unnamed[(expr) ? 1 : 0]; }
```

现在如果你这样写：

```

template <class To, class From>
To safe_reinterpret_cast(From from)
{
    STATIC_CHECK(sizeof(From) <= sizeof(To));
    return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);

```

而如果在你的系统中，指针大小大于字符，编译器会抱怨你“正试着产生一个长度为零的 **array**”。

问题是，你收到的错误消息无法表达正确信息。“无法产生一个长度为零的 **array**”这句话无法暗示“**char** 型别用来持有一个指针实在太小了”。供应“可定制、可移植的错误消息”是很困难的一件事。错误消息之间并没有什么必须遵循的规则，端视编译器而定。例如，如果错误消

⁶ 有了 `safe_reinterpret_cast`，在大多数机器上的确如此，但不能完全保证。

息指出一个未定义变量，则该变量名称不一定得出现在错误消息里头。

较好的解法是依赖一个名称带有意义的 `template`（因为，很幸运地，编译器会在错误消息中指出 `template` 名称）：

```
template<bool> struct CompileTimeError;
template<> struct CompileTimeError<true> {};

#define STATIC_CHECK(expr) \
    (CompileTimeError<(expr) != 0>())
```

`CompileTimeError` 需要一个非型别参数（一个 `bool` 常数），而且它只针对 `true` 有所定义。如果你试着具现化 `CompileTimeError<false>`，编译器会发出 "Undefined specialization `CompileTimeError<false>`" 消息。这个消息比错误消息好，因为它是我们故意制造的，不是编译器或程序的臭虫。

当然这其中还有很多改善空间。如何定制错误消息？我的想法是传入一个额外引数给 `STATIC_CHECK`，并让它在错误消息中出现。唯一的缺点是这个定制消息必须是合法的 C++ 标识符（不能间杂空白、不能以数字开头...）。这个想法引出了一个改良版 `CompileTimeError`，如下所示。此后 `CompileTimeError` 之名不再适用，改为 `CompileTimeChecker` 更具意义：

```
template<bool> struct CompileTimeChecker
{
    CompileTimeChecker(...);    // 译注：这是 C/C++ 支持的非定量任意参数表
};
template<> struct CompileTimeChecker<false> { };
#define STATIC_CHECK(expr, msg) \
    { \
        class ERROR_##msg {}; \        // 译注：## 是个罕被使用的 C++ 操作符
        (void)sizeof(CompileTimeChecker<(expr)>(ERROR_##msg())); \
    }
```

假设 `sizeof(char) < sizeof(void*)`（注意，C++ *Standard* 并不保证这一定为真）。让我们看看当你写出下面这段代码，会发生什么事：

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
    STATIC_CHECK(sizeof(From) <= sizeof(To),
        Destination_Type_Too_Narrow);
    return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);
```

宏被处理完毕后，上述的 `safe_reinterpret_cast` 会被展开成下列样子：

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
    {
        class ERROR_Destination_Type_Too_Narrow {};
        (void)sizeof(
            CompileTimeChecker<(sizeof(From) <= sizeof(To))>{
                ERROR_Destination_Type_Too_Narrow()});
    }
    return reinterpret_cast<To>(from);
}
```

这段程序定义了一个名为 `ERROR_Destination_Type_Too_Narrow` 的 local class，那是一个空类，然后生成一个型别为 `CompileTimeChecker<(sizeof(From) <= sizeof(To))>` 的暂时对象，并以一个型别为 `ERROR_Destination_Type_Too_Narrow` 的暂时对象加以初始化。最终，`sizeof` 会测量出这个对象的大小。

这是个小技巧。`CompileTimeChecker<true>` 这个特化体有一个可接受任何参数的构造函数；它是一个“参数表为省略符 (ellipsis)”的函数。这意味着如果编译期的表达式评估结果为 `true`，这段代码就有效。如果大小比较结果为 `false`，就会有编译期错误发生：因为编译器找不到将 `ERROR_Destination_Type_Too_Narrow` 转成 `CompileTimeChecker<false>` 的方法。最棒的是编译器能够输出如下正确消息：“Error: Cannot convert `ERROR_Destination_Type_Too_Narrow` to `CompileTimeChecker<false>`”，这真是太棒了！

2.2 Partial Template Specialization (模板偏特化)

Partial template specialization 让你在 template 的所有可能实体中特化出一组子集。让我们先扼要解释 template specialization。如果你有这样一个 class template，名为 `Widget`：

```
template <class Window, class Controller>
class Widget
{
    ... generic implementation ...
};
```

你可以像下面这样明白加以特化：

```
template <>
class Widget<ModalDialog, MyController>
{
    ... specialized implementation ...
};
```

其中 `ModalDialog` 和 `MyController` 是你另外定义的 classes。

有了这个 `Widget` 特化定义之后，如果你定义 `Widget<ModalDialog, MyController>` 对象，编译器就使用上述定义，如果你定义其他泛型对象，编译器就使用原本的泛型定义。

然而有时候你也许想要针对任意 `Window` 并搭配一个特定的 `MyController` 来特化 `Widget`。这时就需要 `Partial Template Specialization` 机制：

```
// Partial specialization of Widget
template <class Window>                // 译注: Window 仍是泛化
class Widget<Window, MyController>    // 译注: MyController 是特化
{
    ... partially specialized implementation ...
};
```

通常在一个 `class template` 偏特化定义中，你只会特化某些 `template` 参数而留下其他泛化参数。当你在程序中具体实现上述 `class template` 时，编译器会试着找出最匹配的定义。这个寻找过程十分复杂精细，允许你以富创意的方式来进行偏特化。例如，假设你有一个 `Button class template`，它有一个 `template` 参数，那么，你不但可以拿任意 `Window` 搭配特定 `MyController` 来特化 `Widget`，还可以拿任意 `Button` 搭配特定 `MyController` 来偏特化 `Widget`：

```
template <class ButtonArg>
class Widget<Button<ButtonArg>, MyController>
{
    ... further specialized implementation ...
};
```

如你所见，偏特化的能力十分令人惊讶。当你具现化一个 `template` 时，编译器会把目前存在的偏特化和全特化 `templates` 作比较，并找出其中最合适者。这样的机制给了我们很大的弹性。不幸的是偏特化机制不能用在函数身上（不论成员函数或非成员函数），这样多少会降低一些你所能做出来的弹性和粒度（*granularity*）。

- 虽然你可以全特化 `class template` 中的成员函数，但你不能偏特化它们。
- 你不能偏特化 `namespace-level (non-member)` 函数。最接近“`namespace-level template functions`”偏特化机制的是函数重载——就实际运用而言，那意味着你对“函数参数”（而非返回值型别或内部所用型别）有很精致的特化能力。例如：

```
template <class T, class U> T Fun(U obj);        // primary template
// template <class U> void Fun(void, U)(U obj); // illegal partial
// specialization
template <class T> T Fun (Window obj);          // legal (overloading)
```

如果没有偏特化，编译器设计者的日子肯定会好过一些，但却对程序开发者造成不好的影响。稍后介绍的一些工具（如 `Int2Type` 和 `Type2Type`）都显现偏特化的极限。本书频繁运用偏特化，`typelist` 的所有设施（第3章）几乎都建立在这个机制上。

2.3 局部类 (Local Classes)

这是一个有趣而少有人知道的 C++ 特性。你可以在函数中定义 `class`，像下面这样：

```
void Fun()
{
    class Local
    {
        ... member variables ...
        ... member function definitions ...
    };
    ... code using Local ...
}
```

不过还是有些限制，`local class` 不能定义 `static` 成员变量，也不能访问 `non-static` 局部变量。`local classes` 令人感兴趣的是，可以在 `template` 函数中被使用。定义于 `template` 函数内的 `local classes` 可以运用函数的 `template` 参数。以下所列代码中有一个 `MakeAdapter` `template function`，可以将某个接口转接为另一个接口。`MakeAdaptery` 在其 `local class` 的协助下实作出一个接口。这个 `local class` 内有泛化型别的成员。

```
class Interface
{
public:
    virtual void Fun() = 0;
    ...
};

template <class T, class P>
Interface* MakeAdapter(const T& obj, const P& arg)
{
    class Local : public Interface
    {
public:
        Local(const T& obj, const P& arg)
            : obj_(obj), arg_(arg) {}
        virtual void Fun()
        {
            obj_.Call(arg_);
        }
private:
        T obj_;
        P arg_;
    };
    return new Local(obj, arg);
}
```

事实证明，任何运用 local classes 的手法，都可以改用“函数外的 template classes”来完成。换言之，并非一定得 local classes 不可。不过 local classes 可以简化实作并提高符号的地域性。

Local classes 倒是有个独特性质：它们是“最后一版”（也即 Java 口中的 final）。外界不能继承一个隐藏于函数内的 class。如果没有 local classes，为了实现 Java final，你必须在编译单元（译注：也就是个别文件）中加上一个无具名的命名空间（namespace）。

我将在第 11 章运用 local classes 产生所谓的“弹簧垫”函数（trampoline functions）。

2.4 常整数映射为型别 (Mapping Integral Constants to Types)

下面是最初由 Alexandrescu (2000b) 提出的一个简单 template，对许多泛型编程手法很有帮助：

```
template <int v>
struct Int2Type
{
    enum { value = v };
};
```

Int2Type 会根据引数所得的不同数值来产生不同型别。这是因为“不同的 template 具现体”本身便是“不同的型别”。因此 Int2Type<0> 不同于 Int2Type<1>，以此类推。用来产生型别的那个数值是一个枚举值（enumerator）。

当你想把常数视同型别时，便可采用上述的 Int2Type。这么一来便可根据编译期计算出来的结果选用不同的函数。实际上你可以运用一个常数达到静态分派（static dispatching）功能。

一般而言，符合下列两个条件便可使用 Int2Type：

- 有必要根据某个编译期常数调用一个或数个不同的函数。
- 有必要在编译期实施“分派”（dispatch）。

如果打算在执行期进行分派（dispatch），可使用 if-else 或 switch 语句。大部分时候其执行期成本都微不足道。然而你还是无法常常那么做，因为 if-else 语句要求每一个分支都得编译成功，即使该条件测试在编译期才知道。困惑了吗？读下去！

假想你设计出一个泛形容器 NiftyContainer，它将元素型别参数化：

```
template <class T> class NiftyContainer
{
    ...
};
```

现在假设 NiftyContainer 内含指针，指向型别为 T 的对象。为了复制 NiftyContainer 里的某个对象，你想调用其 copy 构造函数（针对 non-polymorphic 型别）或虚函数 Clone()（针对 polymorphic 型别）。你以一个 boolean template 参数取得使用者所提供的信息：

```

template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    void DoSomething()
    {
        T* pSomeObj = ...;
        if (isPolymorphic)
        {
            T* pNewObj = pSomeObj->Clone();
            ... polymorphic algorithm ... (多态算法)
        }
        else
        {
            T* pNewObj = new T(*pSomeObj); // 译注: copy 构造函数
            ... non-polymorphic algorithm ... (非多态算法)
        }
    }
};

```

问题是，编译器不会让你侥幸成功。如果多态算法使用 `pObj->Clone()`，那么面对任何一个未曾定义成员函数 `Clone()` 之型别，`NiftyContainer::DoSomething()` 都无法编译成功。虽然编译期间很容易知道哪一条分支会被执行起来，但这和编译器无关，因为即使优化工具可以评估出哪一条分支不会被执行，编译器还是会勤劳地编译每个分支。如果你试着调用 `NiftyContainer<int,false>` 的 `DoSomething()`，编译器会停在 `pObj->Clone()` 处并说“嗨，不行唷”。

上述的 `non-polymorphic` 部分也有可能编译失败。如果 `T` 是个 `polymorphic` 型别，而上述的 `non-polymorphic` 程序分支想做 `new T(*pObj)` 动作，这样也有可能编译失败。举个实例，如果 `T` 借着“把 `copy` 构造函数置于 `private` 区域以产生隐藏效果”，就像一个有良好设计的 `polymorphic class` 那样，那么便有可能出现上述的失败情况。

如果编译器不去理会那个不可能被执行的代码就好了，然而目前情况下是不可能的。甚么才是令人满意的解决方案呢？

事实证明有很多解法，而 `Int2Type` 提供了一个特别明确的方案。它可以把 `isPolymorphic` 这个型别的 `true` 和 `false` 转换成两个可资区别的不同型别，然后在程序中便可以运用 `Int2Type<isPolymorphic>` 进行函数重载。瞧，可不是吗！

```

template <typename T, bool isPolymorphic>
class NiftyContainer
{
private:
    void DoSomething(T* pObj, Int2Type<true>)
    {
        T* pNewObj = pObj->Clone();
        ... polymorphic algorithm ...
    }
}

```



```

void DoSomething(T* pObj, Int2Type<false>)
{
    T* pNewObj = new T(*pObj);
    ... nonpolymorphic algorithm ...
}
public:
void DoSomething(T* pObj)
{
    DoSomething(pObj, Int2Type<isPolymorphic>());
}
};

```

`Int2Type` 是一个用来“将数值转换为型别”的方便手法。有了它，你便可以将该型别的一个暂时对象传给一个重载函数(overloaded function)，后者实现必要的算法。(译注：这种手法在 STL 中亦有大量实现，唯形式略有不同；详见 STL 源码，或《STL 源码剖析》by 侯捷)

这个小技巧之所以有效，最主要的原因是，编译器并不会去编译一个未被用到的 `template` 函数，只会对它做文法检查。至于此技巧之所以有用，则是因为在 `template` 代码中大部分情形下你需要在编译期做流程分派(dispatch)动作。

你会在 Loki 的数个地方看到 `Int2Type` 的运用，尤其是本书第 11 章：Multimethods。在那儿，`template class` 是一个双分派(double-dispatch)引擎，运用 `bool` `template` 参数决定是否要支持对称性分派(symmetric dispatch)。

2.5 型别对型别的映射(Type-to-Type Mapping)

就如 2.2 节所说，并不存在 `template` 函数的偏特化。然而偶尔我们需要模拟出类似机制。试想下面的程序：

```

template <class T, class U>
T* Create(const U& arg)
{
    return new T(arg);
}

```

`Create()` 会将其参数传给构造函数，用以产生一个新对象。

现在假设你的程序有个规则：`widget` 对象是你碰触不到的老旧代码，它需要两个引数才能构造出对象来，第二引数固定为 `-1`。`widget` 派生类则没有这个问题。

现在你该如何特化 `Create()`，使它能够独特地处理 `widget` 呢？一个明显的方案是另写出一个 `CreateWidget()` 来专门处理。但这么一来你就没有一个统一的接口用来生成 `widgets` 和其派生对象。这会使得 `Create()` 在任何泛型程序中不再有用。

由于你无法偏特化一个函数，因此无法写出下面这样的代码：

```

// Illegal code — don't try this at home
template <class U>
Widget* Create<Widget, U>(const U& arg)

```

```
{
    return new Widget(arg, -1);
}
```

由于函数缺乏偏特化机制，因此（再一次地）你只有一样工具可用：多载化（重载）机制。我们可以传入一个型别为 `T` 的暂时对象，并以此进行重载：

```
template <class T, class U>
T* Create(const U& arg, T /* dummy */)
{
    return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Widget /* dummy */)
{
    return new Widget(arg, -1);
}
```

这种解法会轻易构造未被使用的复杂对象，造成额外开销。我们需要一个轻量级机制来传递“型别 `T` 的信息”到 `Create()` 中。这正是 `Type2Type` 扮演的角色，它是一个型别代表物，一个可以让你传给重载函数的轻量级 ID。 `Type2Type` 定义如下：

```
template <typename T>
struct Type2Type
{
    typedef T OriginalType;
};
```

它没有任何数值，但其不同型别却足以区分各个 `Type2Type` 实体，这正是我们所要的。现在你可以这么写：

```
// An implementation of Create relying on overloading
// and Type2Type
template <class T, class U>
T* Create(const U& arg, Type2Type<T>)
{
    return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Type2Type<Widget>)
{
    return new Widget(arg, -1);
}
// Use Create()
String* pStr = Create("Hello", Type2Type<String>{});
Widget* pW = Create(100, Type2Type<Widget>{});
```

`Create()` 的第二参数只是用来选择适当的重载函数, 现在你可以令各种 `Type2Type` 实体对应于你的程序中的各种型别, 并根据不同的 `Type2Type` 实体来特化 `Create()`。

2.6 型别选择 (Type Selection)

有时候, 泛型程序需要根据一个 `boolean` 变量来选择某个型别或另一型别。

在 2.4 节讨论的 `NiftyContainer` 例子中, 你也许会以一个 `std::vector` 作为后端存储结构。很显然, 面对 `polymorphic` (多态) 型别, 你不能存储其对象实体, 必须存储其指针。但如果面对的是 `non-polymorphic` (非多态) 型别, 你可以存储其实体, 因为这样比较有效率。

在你的 `class template` 中:

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
};
```

你需要存放一个 `vector<T*>` (如果 `isPolymorphic` 为 `true`) 或 `vector<T>` (如果 `isPolymorphic` 为 `false`)。根本而言, 你需要根据 `isPolymorphic` 来决定将 `ValueType` 定义为 `T*` 或 `T`。你可以使用 `traits class template` (Alexandrescu 2000a) 如下:

```
template <typename T, bool isPolymorphic>
struct NiftyContainerValueTraits
{
    typedef T* ValueType;
};
template <typename T>
struct NiftyContainerValueTraits<T, false>
{
    typedef T ValueType;
};
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    typedef NiftyContainerValueTraits<T, isPolymorphic> Traits;
    typedef typename Traits::ValueType ValueType;
};
```

这样的做法其实笨拙难用, 此外它也无法扩充: 针对不同的型别的选择, 你都必须定义出专属的 `traits class template`。

Loki 提供的 `Select class template` 可使型别的选择立时可用。它采用偏特化机制 (`partial template specialization`) :

```

template <bool flag, typename T, typename U>
struct Select
{
    typedef T Result;
};
template <typename T, typename U>
struct Select<false, T, U>
{
    typedef U Result;
};

```

其运作方式是：如果 `flag` 为 `true`，编译器会使用第一份泛型定义，因此 `Result` 会被定义成 `T`。如果 `flag` 为 `false`，那么偏特化机制会进场运作，于是 `Result` 被定义成 `U`。

现在你可以更方便地定义 `NiftyContainer::ValueType` 了：

```

template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    typedef Select<isPolymorphic, T*, T>::Result    ValueType;
    ...
};

```

2.7 编译期间侦测可转换性 (Convertibility) 和继承性 (Inheritance)

实作 `template functions` 和 `template classes` 时我常常发现一个问题：面对两个陌生的型别 `T` 和 `U`，如何知道 `U` 是否继承自 `T` 呢？在编译期间发掘这样的关系，实在是实作泛型程序库的一个优化关键。在泛型函数中，如果你确知某个 `class` 实作有某个接口，你便可以采用某个最佳算法。在编译期发现这样的关系，意味着不必使用 `dynamic_cast`——它会耗损执行期效率。

发掘继承关系，靠的是一个用来侦测可转换性 (convertibility) 的更一般化机制。这里我们面临更一般化的问题：如何测知任意型别 `T` 是否可以自动转换为型别 `U`？

有个方案可以解决问题，并且只需仰赖 `sizeof`。`sizeof` 有着惊人的威力：你可以把 `sizeof` 用在任何表达式 (expression) 身上，不论后者有多复杂。`sizeof` 会直接传回大小，不需拖到执行期才评估。这意味着 `sizeof` 可以感知重载 (overloading)、模板具现 (template instantiation)、转换规则 (conversion rules)，或任何可发生于 C++ 表达式身上的机制。事实上 `sizeof` 背后隐藏了一个“用以推导表达式型别”的完整设施。最终 `sizeof` 会丢弃表达式并传回其大小⁷。

“侦测转换能力”的想法是：合并运用 `sizeof` 和重载函数。我们提供两个重载函数：其中一

⁷目前正有一份提案，准备为 C++ 语言加入 `typeof` 操作符，它会传回一个表达式的型别。有了这个 `typeof` 操作符，泛型程序将会更好写，也更易被了解。Gnu C++ 已实作出 `typeof` 作为语言扩充功能。很显然 `typeof` 和 `sizeof` 拥有共同的后端实作，因为 `sizeof` 也需要判断型别。

一个接受 U (U 型别代表目前讨论中的转换目标)，另一个接受“任何其他型别”。我们以型别 T 的暂时对象来调用这些重载函数，而“ T 是否可转换为 U ”正是我们想知道的。如果接受 U 的那个函数被调用，我们就知道 T 可转换为 U ；否则 T 便无法转换为 U 。为了知道哪一个函数被调用，我们对这两个重载函数安排大小不同的返回型别，并以 `sizeof` 来区分其大小。型别本身无关紧要，重要的是其大小必须不同。

让我们先建立两个不同大小的型别。很显然 `char` 和 `long double` 的大小不同，不过 C++ 标准规格书并未保证此事，所以我想到一个极其简单的做法：

```
typedef char Small;
class Big { char dummy[2]; };
```

根据定义，`sizeof(Small)` 是 1。而 `Big` 的大小肯定比 1 还大，这正是我们所需要的保证。

接下来需要两个重载函数，其一如先前所说，接受一个 U 对象并传回一个 `Small` 对象：

```
Small Test(U);
```

但接下来，我该如何写出一个可接受任何其他种对象的函数呢？`template` 并非解决之道，因为 `template` 总是要求最佳匹配条件，因而遮蔽了转换动作。我需要一个“比自动转换稍差”的匹配，也就是说我需要一个“唯有在自动转换缺席情况下”才会雀屏中选的转换。我很快看了一下施行于函数调用的各种转换规则，然后发现所谓的“省略符比对”准则，那是最坏的情况了，位于整个列表的最底端。于是我写出这样一个函数：

```
Big Test(...);
```

（调用一个带有省略符的函数并传入一个 C++ 对象，无人知道结果会如何。不过没关系，我们并不真正调用这个函数，它甚至没被实作出来。还记得吗，`sizeof` 并不对其引数求值）

现在我们传一个 T 对象给 `Test()`，并将 `sizeof` 施行于其传回值身上：

```
const bool convExists = sizeof(Test(T())) == sizeof(Small);
```

就是这样！`Test()` 会取得一个 default 构造对象 `T()`，然后 `sizeof` 会取得这一表达式结果的大小，可能是 `sizeof(Small)` 或 `sizeof(Big)`，取决于编译器是否找到转换方式。

这里还有一个小问题。万一 T 让自己的 default 构造函数为 `private`，那么 `T()` 会编译失败，我们所有的舞台支架也将倒塌。庆幸的是有一个简单解法：以一个“稻草人函数” (strawman function) 传回一个 T 对象。还记得吗，我们处于 `sizeof` 的神奇世界中，并不会真有任何表达式被求值 (evaluated)。本例之中编译器高兴，我们也高兴。

```
T MakeT(); // not implemented
const bool convExists = sizeof(Test(MakeT())) == sizeof(Small);
```

(顺便提一下, 像 `MakeT()` 和 `Test()` 这样的函数, 你能在它们身上做多少事情? 它们不只没做任何事情, 甚至根本不真正存在。这不是很好玩吗?)

现在让它运作, 把所有东西以 `class template` 包装起来, 隐藏“型别推导”的所有细节, 只暴露结果:

```
template <class T, class U>
class Conversion
{
    typedef char Small;
    class Big { char dummy[2]; };
    static Small Test(U);
    static Big Test(...);
    static T MakeT();
public:
    enum { exists =
        sizeof(Test(MakeT())) == sizeof(Small) };
};
```

下面的代码用来测试上述的 `Conversion class template`:

```
int main()
{
    using namespace std;
    cout << Conversion<double, int>::exists << ' '
        << Conversion<char, char*>::exists << ' '
        << Conversion<size_t, vector<int>>::exists << ' ';
}
```

这个小程序会印出 "100"。注意, 虽然 `std::vector` 实作出一个接受 `size_t` 引数的构造函数, 但上述转换测试却传回 0, 因为该构造函数是 `explicit` (译注: `explicit` 构造函数无法担任转换函数)。

我们在 `Conversion` 中更实作出两个常数:

- `exists2Way`, 表示 `T` 和 `U` 之间是否可以双向转换。例如 `int` 和 `double` 可以双向转换。使用者自定义型别也可以实作出这样的转换。
- `sameType`: 如果 `T` 和 `U` 是相同型别, 这个值便为 `true`。

```
template <class T, class U>
class Conversion
{
    ... as above ...
    enum { exists2Way = exists &&
        Conversion<U, T>::exists };
    enum { sameType = false };
};
```

我们也可以通过 `Conversion` 的偏特化 (partial specialization) 来实作出 `sameType`:

```
template <class T>
class Conversion<T, T>
```

```
{
public:
    enum { exists = 1, exists2Way = 1, sameType = 1 };
};
```

最后，让我们回头看看，有了 `Conversion` 的帮助，要决定两个 `classes` 之间是否存在继承关系，变得很容易：

```
#define SUPERSUBCLASS(T, U) \
    (Conversion<const U*, const T*>::exists && \
     !Conversion<const T*, const void*>::sameType)
```

如果 `U` 是 `public` 继承自 `T`，或 `T` 和 `U` 是同一型别，那么 `SUPERSUBCLASS(T,U)` 会传回 `true`。当 `SUPERSUBCLASS(T,U)` 对 `const U*` 和 `const T*` 作“可转换性”评估时，只有三种情况下 `const U*` 可以隐式转换为 `const T*`：

1. `T` 和 `U` 是同一种型别。
2. `T` 是 `U` 的一个 `unambiguous`（不模棱两可的、非歧义的）`public base`。
3. `T` 是 `void`。

第三种情况可以在前述第二次测试中解决掉。如果把第一种情况（`T` 和 `U` 是同一型别）视为 **is-a** 的退化，实作时会很有用，因为实用场合中你常常可以将某个 `class` 视为它自己的 `superclass`。如果你需要更严谨的测试，可以这么写：

```
#define SUPERSUBCLASS_STRICT(T, U) \
    (SUPERSUBCLASS(T, U) && \
     !Conversion<const T, const U>::sameType)
```

为何这些代码都加上 `const` 饰词？原因是我们不希望因 `const` 而导致转型失败。如果 `template` 代码实施 `const` 两次（对一个已经是 `const` 的型别而言），第二个 `const` 会被忽略。简单地讲，藉由在 `SUPERSUBCLASS` 中使用 `const`，我们得以更安全一些。

为什么选用 `SUPERSUBCLASS` 而不是更可爱的名称，如 `BASE_OF` 或 `INHERITS` 之类？这是基于一个非常实际的理由。一开始 `Loki` 对它的命名是 `INHERITS`，但是当 `INHERITS(T,U)` 运作时，出现了一个问题：它说的是 `T` 继承 `U` 或相反呢？改名为 `SUPERSUBCLASS(T,U)` 之后，谁先谁后就变得很清楚。

2.8 type_info 的一个外覆类 (Wrapper)

标准 C++ 提供了一个 `std::type_info` `class`，使你能够于执行期间查询对象型别。通常 `type_info` 必须和 `typeid` 操作符并用，后者会传回一个 `reference`，指向一个 `type_info` 对象：

```
void Fun(Base* pObj)
{
    // Compare the two type_info objects corresponding
    // to the type of *pObj and Derived
```

```

    if (typeid(*pObj) == typeid(Derived))
    {
        ... aha, pObj actually points to a Derived object ...
    }
    ...
}

```

除了支持比较运算 `operator==` 和 `operator!=` 外, `type_info` 还提供如下两个函数:

- `name()`, 传回一个 `const char*` 代表型别名称, 但是并无标准方法可以将 `class` 名称对应于字符串, 所以你不应该期望 `typeid(Widget)` 会传回 `"Widget"`。一个可资遵循 (但不至于获奖) 的做法是让 `type_info::name()` 对所有型别都传回空字符串。
- `before()`, 带来 `type_info` 对象的次序关系。程序员可运用 `type_info::before()` 对 `type_info` 对象建立索引。

不幸的是, 这些好用的 `type_info` 功能被包装得难以发挥。`type_info` 关闭了 `copy` 构造函数和 `assignment` 操作符, 使我们不可能存储 `type_info` 对象, 但你可以存储一个指向 `type_info` 对象的指针。`typeid` 传回的对象采用 `static` 存储方式, 所以你不必担心其寿命问题, 只需担心指针间的辨识就行了。

C++ *Standard* 并不保证每次调用 (例如) `typeid(int)`, 会传回 “指向同一个 `type_info` 对象” 的 `reference`。如此一来, 你就无法比较 “指向 `type_info` 对象” 的指针了。因此, 你应该做的是存储 `type_info` 对象, 并且以 `type_info::operator==` 施行于提领后的指针上。

如果你想对 `type_info` 对象排序, 你必须再一次存储 “指向 `type_info` 对象” 的指针, 此时必须使用成员函数 `before()`。因此, 如果你想要使用 STL 的带序容器 (`ordered containers`), 必须写出一个小小的仿函数 (`functor`) 并处理指针。

以上这些已经够笨拙到足以让我们委派一个外覆类 (`wrapper class`) 来包装 `type_info` 了, 它应该存储 “指向 `type_info` 对象” 的指针, 并提供下列功能:

- `type_info` 的所有成员函数。
- `value` 语义 (译注: 相对于 `reference` 语义), 也就是 `public copy` 构造函数和 `public assignment` 操作符。
- 定义出 `operator<` 和 `operator==`, 使比较动作更完整。

Loki 已经实作出如此好用的外覆类, 名为 `TypeInfo`, 概要如下:

```

class TypeInfo
{
public:
    // Constructors/destructors
    TypeInfo();    // needed for containers
    TypeInfo(const std::type_info&);

```



```

TypeInfo(const TypeInfo&);
TypeInfo& operator=(const TypeInfo&);
// Compatibility functions
bool before(const TypeInfo&) const;
const char* name() const;
private:
    const std::type_info* pInfo_;
};
// Comparison operators
bool operator==(const TypeInfo&, const TypeInfo&);
bool operator!=(const TypeInfo&, const TypeInfo&);
bool operator<(const TypeInfo&, const TypeInfo&);
bool operator<=(const TypeInfo&, const TypeInfo&);
bool operator>(const TypeInfo&, const TypeInfo&);
bool operator>=(const TypeInfo&, const TypeInfo&);

```

由于其中的“转换构造函数”接受一个 `std::type_info` 参数，所以你可以拿一个 `TypeInfo` 对象和一个 `std::type_info` 对象来比较，像这样：

```

void Fun(Base* pObj)
{
    TypeInfo info = typeid(Derived);
    ...
    if (typeid(*pObj) == info)
    {
        ... pBase actually points to a Derived object ...
    }
    ...
}

```

许多情况下，`TypeInfo` 对象的复制和比较能力是很重要的。第 8 章的 `cloning factory` 和第 11 章的 `double-dispatch` 引擎都把 `TypeInfo` 运用得很好。

2.9 NullType 和 EmptyType

Loki 定义了两个非常简单的型别：`NullType` 和 `EmptyType`。你可以拿它们当做型别计算时的某种边界标记。

`NullType` 是一个只有声明而无定义的 `class`：

```
Class NullType;    // no definition
```

你不能生成一个 `NullType` 对象，它只被用来表示“我不是个令人感兴趣的型别”。2.10 节把 `NullType` 用在有语法需求却无语义概念的地方（例如“`int` 指的是什么型别”）。第 3 章的 `typelist` 以 `NullType` 标记 `typelist` 的末端，并用以传回“找不到型别”这一消息。

第二个辅助型别是 `EmptyType`。和你想的一样，`EmptyType` 定义如下：

```
struct EmptyType {};
```

这是一个可被继承的合法型别，而且你可以传递 `EmptyType` 对象。你可以把这个轻量级型别视为 `template` 的缺省（可不理会的）参数型别。第3章的 `typelist` 就是这样用它的。

2.10 Type Traits

Traits 是一种“可于编译期根据型别作判断”的泛型技术，很像你在执行期根据数值进行判断一样（Alexandrescu 2000a）。众所皆知，加上一个间接层便可解决很多工程问题，`trait` 让你得以在“型别确立当时”以外的其他地点做出与型别相关的判断。这会让最终的代码变得比较干净，更具可读性，而且更好维护。

通常，当你的泛型程序需要时，你会写出自己的 `trait templates` 和 `trait classes`。然而某些 `traits` 可应用于任何型别，它们可以帮助泛型程序员根据型别特性修改出适当的泛型代码。

举个例子。假设你想实作 `copying` 算法：

```
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
    for (; first != last; ++first, ++result)
        *result = *first;
}
```

理论上并不需要实作这样的算法，因为这和 `std::copy()` 的功能重复了，但也许需要针对某些型别，对这个算法进行特化。

假设你要在一台多处理器（`multiprocessor`）机器上发展程序，它已有一个非常快的内建函数 `BitBlast()`，而你希望尽可能发挥该函数的好处。

```
// Prototype of BitBlast in "SIMD_Primitives.h"
void BitBlast(const void* src, void* dest, size_t bytes);
```

当然，`BitBlast()` 只对基本型别和简朴的旧式结构（译注：意指类似 C `struct` 这样的东西）运作。你不能把 `BitBlast()` 用于拥有“`nontrivial copy` 构造函数”的型别上。现在，你希望尽可能利用 `BitBlast()` 来实作前述的 `Copy` 算法，并希望退却至较为一般、谨慎的做法，希望适用于各种精巧型别。如此一来，拷贝动作面对基本型别时便会“自动”快速执行。

你需要做两个测试（译注：以下两变量是前述 `Copy()` 的参数）：

- `InIt` 和 `OutIt` 是一般指针吗（相对于较华丽、需高度技巧的迭代器（`iterator`）而言）？
- `InIt` 和 `OutIt` 所指的型别可以进行 `bitwise copy`（位逐一拷贝）吗？

如果你能够在编译期找出这两个问题的答案，而且答案都是 `yes`，那么便可以采用 `BitBlast()`，否则就只能以一般循环来完成。

Type traits 有助于解决这样的问题。本章的 `type traits` 归功于 Boost（一个 C++ 程序库）实作出来的许多 `type traits`。

2.10.1 实作出 Pointer Traits

Loki 定义了一个 class template `TypeTraits`，收纳很多泛型 traits。`TypeTraits` 内部使用 `template specialization`（模板特化）并将结果显露出来。

大部分 type traits 的实作都需倚赖 `template` 的全特化或偏特化（2.2 节）。例如下面这段代码用来判断型别 `T` 是否为指针：

```
template <typename T>
class TypeTraits
{
private:
    template <class U> struct PointerTraits
    {
        enum { result = false };
        typedef NullType PointeeType;
    };
    template <class U> struct PointerTraits<U*>
    {
        enum { result = true };
        typedef U PointeeType;
    };
public:
    enum { isPointer = PointerTraits<T>::result };
    typedef PointerTraits<T>::PointeeType PointeeType;
    ...
};
```

其中第一个 class template 导入 `PointerTraits` 的定义，其意义是：“`T` 不是指针，且所谓被指型别（`PointeeType`）不能拿来运用”。回忆 2.9 节所说，是的，`NullType` 只是一种用于“不能被使用”情况下的占位型别（placeholder type）。

第二个 `PointerTraits`（粗体那一行）是上一个 `PointerTraits` 的偏特化，是一个“针对任意指针型别”的特化体。对任意指针而言，这个特化体比其泛型版本更适合作为候选者。因此，如果面对的是指针，就进入这个特化体运作，因此 `result` 是 `true`，此外 `PointeeType` 会被适当定义。

现在请观察 `std::vector::iterator` 的实作内容，它是一个简朴指针还是一个精巧型别？

```
int main()
{
    const bool
        iterIsPtr = TypeTraits<vector<int>::iterator>::isPointer;
    cout << "vector<int>::iterator is " <<
        iterIsPtr ? "fast" : "smart" << '\n';
}
```

同理，`TypeTraits` 也实作出一个 `isReference` 常数和 `ReferencedType` 型别。对于一个 `reference type` `T` 而言，`ReferencedType` 代表“`T` 所参考（指向）”的型别；如果 `T` 是个直率的型别（译注：亦即 non-reference），那么 `ReferencedType` 就是 `T` 自己。

如欲侦测 `pointers to members` (参见第5章相关说明), 情况有点不同。我们需要特化如下:

```
template <typename T>
class TypeTraits
{
private:
    template <class U> struct PToMTraits
    {
        enum { result = false };
    };
    template <class U, class V>
    struct PToMTraits<U V::*>
    {
        enum { result = true };
    };
public:
    enum { isMemberPointer = PToMTraits<T>::result };
    ...
};
```

2.10.2 侦测基本型别 (fundamental types) (译注: 或称 primitive types)

`TypeTraits<T>` 实作出一个编译期常数 `isStdFundamental`, 用来表示 `T` 是否为基本型别。标准基本型别包括 `void` 和所有数值型别 (分为浮点数和整数两大集团)。`TypeTraits` 定义了一些常数用来表示某个型别属于哪一分类。

让我未雨绸缪一下, 先说明第3章的 `typelists` 的神奇魔法。它可以侦测某个型别是否隶属某一组型别。此刻你唯一应该知道的是:

```
TL::IndexOf<T, TYPELIST_nn (以逗号隔开的 type list) >::value
```

其中的 `nn` 代表 `type list` 中有多少个型别。上式会传回 `T` 在 `type list` 中的位置 (从零起算); 如果 `T` 不在其中就传回 `-1`。举个例子, 如果 `TL::IndexOf<T, TYPELIST_4(signed char, short int, int, long int) >::value` 的值大于或等于零, 表示 `T` 是个带正负号的整数。

`TypeTraits` 针对基本型别, 有如下定义:

```
template <typename T>
class TypeTraits
{
    ... as above ...
public:
    typedef TYPELIST_4(
        unsigned char, unsigned short int,
        unsigned int, unsigned long int)
        UnsignedInts;
    typedef TYPELIST_4(signed char, short int, int, long int)
        SignedInts;
```

```

typedef TYPELIST_3(bool, char, wchar_t) OtherInts;
typedef TYPELIST_3(float, double, long double) Floats;
enum { isStdUnsignedInt =
    TL::IndexOf<T, UnsignedInts>::value >= 0 };
enum { isStdSignedInt = TL::IndexOf<T, SignedInts>::value >= 0 };
enum { isStdIntegral = isStdUnsignedInt || isStdSignedInt ||
    TL::IndexOf<T, OtherInts>::value >= 0 };
enum { isStdFloat = TL::IndexOf<T, Floats>::value >= 0 };
enum { isStdArith = isStdIntegral || isStdFloat };
enum { isStdFundamental = isStdArith || isStdFloat ||
    Conversion<T, void>::sameType };
...
};

```

使用 `typelists` 和 `TL::IndexOf`，你会获得一种能力，可以很快推论出型别信息，不需要多次撰写 `template` 特化体。如果你忍不住现在就想研究 `typelists` 和 `TL::Find` 的细节，可以先跳过去看看第 3 章，但是别忘了回到这里喔。

“基本型别侦测动作”的真正实作品比这里所说的更为完整而严谨，并允许厂商自行提供扩充型别（例如 `int64` 或 `long long`）。

2.10.3 优化的参数型别

在泛型代码中，你常常需要回答下列问题：任意给定一个型别 `T`，什么是“将 `T` 对象传入函数当作参数”的最有效做法？一般而言，最有效的方法是在传入一个精巧型别时采用 `by reference` 传递方式，面对纯量型别时采用 `by value` 传递方式。纯量型别由前述之数值型别、枚举型别（`enums`）、指针、指向成员之指针组成。对精巧型别而言，你应该避免额外暂时对象带来的额外开销（那会带来构造函数和析构函数的额外调用动作）；对纯量型别而言，你应该避免 `reference` 带来的间接性所造成的额外开销。

有一个细节必须谨慎处理，那就是 C++ 不允许 `references to references`。因此，如果 `T` 已经是个 `reference`，千万别对它再加一层 `reference`。

我对函数调用的最佳参数型别作了一些分析之后，产生以下算法。让我们把即将寻求的参数型别称为 `ParameterType`：

如果 `T` 是某型别的 `reference`，则 `ParameterType` 就是 `T`，因为 C++ 不允许 `references to references`。否则：

如果 `T` 是个纯量型别（`int`、`float` 等等），`ParameterType` 便是 `T`。因为基本型别的最佳传递方式是 `by value`。

否则 `ParameterType` 将是 `T&`，因为一般“非基本型别”的最佳传递方式是 `by reference`。

这个算法的一个重要成就是它可以免除 `reference-to-reference` 的错误——这个错误可能出现在你结合标准程序库的 `bind2nd` 和 `mem_fun` 时发生（译注：二者都是配接器，`adapters`）。

如果想以我们目前手上的技术，加上先前定义的 `referencedType` 和 `isPrimitive`，实作出 `TypeTraits::ParameterType` 是很容易的：

```

template <typename T>
class TypeTraits
{
    ... as above ...
public:
    typedef Select<isStdArith || isPointer || isMemberPointer,
                  T, ReferencedType&>::Result
        ParameterType;
};

```

不幸的是这个方案如果遇上“以 *by value* 方式传递枚举型别 (enums)”会失败，因为目前已知的任何方法都无法侦测出某个型别是否为 `enum`。

本书第5章的 `Functor class template` 使用了上述的 `TypeTraits::ParameterType`。

2.10.4 卸除饰词 (Stripping Qualifiers)

已知某个型别 `T`，你可以轻易藉由 `const T` 取得其常数版兄弟。然而，相反的动作（卸除某个型别的 `const`）就有点难度了。同理，有时候你会想要卸除某个型别的 `volatile` 饰词。

举个例子。考虑建立一个名为 `SmartPtr` 的 smart pointer class（第7章会详尽讨论这玩意儿）。虽然你允许使用者产生一个 smart pointer to `const` object，例如 `SmartPtr<const widget>`，但在内部你还是需要改变指针，令它指向 `widget`。这种情况下你需要在 `SmartPtr` 内部维护一个由 `const widget` 转来的 `widget` 对象。

实作一个“`const` 卸除器”很容易，我们再次运用 `partial template specialization`（模板偏特化）：

```

template <typename T>
class TypeTraits
{
    ... as above ...
private:
    template <class U> struct UnConst
    {
        typedef U Result;
    };
    template <class U> struct UnConst<const U>
    {
        typedef U Result;
    };
public:
    typedef UnConst<T>::Result NonConstType;
};

```

2.10.5 运用 TypeTraits

`TypeTraits` 可以协助你做出很多有趣的事情。其一是你现在可以结合本章提供的一些技术来使用 `BitBlast()` (2.10 节) 并实作出 `Copy()`。你可以利用 `TypeTraits` 判断两个迭代器 (iterators) 的型别信息，并以 `Int2Type` 决定调用（分派至）`BitBlast()` 或典型的 `copy` 函数。

```

enum CopyAlgoSelector { Conservative, Fast };

// Conservative routine-works for any type
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Conservative>)
{
    for (; first != last; ++first, ++result)
        *result = *first;
}

// Fast routine-works only for pointers to raw data
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Fast>)
{
    const size_t n = last--first;
    BitBlast(first, result, n * sizeof(*first));
    return result + n;
}

template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
    typedef TypeTraits<InIt>::PointeeType SrcPointee;
    typedef TypeTraits<OutIt>::PointeeType DestPointee;
    enum { copyAlgo =
        TypeTraits<InIt>::isPointer &&
        TypeTraits<OutIt>::isPointer &&
        TypeTraits<SrcPointee>::isStdFundamental &&
        TypeTraits<DestPointee>::isStdFundamental &&
        sizeof(SrcPointee) == sizeof(DestPointee) ? Fast : Conservative };

    return CopyImpl(first, last, result, Int2Type<copyAlgo>);
}

```

虽然 `Copy()` 本身没有做很多事情，不过最有趣的其实就在这里。`enum copyAlgo` 会自动选择某个算法，逻辑如下：如果两个迭代器（iterators）都是指针，而且都指向基本型别，而且所指型别的大小一样，那么就使用 `BitBlast()`。最后一个条件是一个有趣的变形，如果你这么做：

```

int* p1 = ...;
int* p2 = ...;
unsigned int* p3 = ...;
Copy(p1, p2, p3);

```

那么 `Copy()` 会调用快速版本，虽然“源端型别”和“目标端型别”并不相同。

这个 `Copy()` 的缺点是它无法加速所有可加速的东西。假设你有一个 C struct，其内除了基本型别的数据外，没有其他任何东西，此即所谓 *plain old data*，或称为 **POD** 结构。C++ Standard 允许对 POD 结构采取 *bitwise copy*（位逐一拷贝）动作，但 `Copy()` 却无法侦测出操作对象是否为 POD，因此它会调用慢速版本。这里你不但需要 `TypeTraits`，还需要“古典的”traits 技法，如下：

```

template <typename T> struct SupportsBitwiseCopy
{
    enum { result = TypeTraits<T>::isStdFundamental };
};
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result, Int2Type<true>)
{
    typedef TypeTraits<InIt>::PointeeType SrcPointee;
    typedef TypeTraits<OutIt>::PointeeType DestPointee;
    enum { useBitBlast =
        TypeTraits<InIt>::isPointer &&
        TypeTraits<OutIt>::isPointer &&
        SupportsBitwiseCopy<SrcPointee>::result &&
        SupportsBitwiseCopy<DestPointee>::result &&
        sizeof(SrcPointee) == sizeof(DestPointee) };
    return CopyImpl(first, last, result, Int2Type<useBitBlast>);
}

```

现在，如欲解除“POD 型别无法运用 `BitBlast()`”的束缚，只需令 `SupportsBitwiseCopy` 针对你的 POD 型别进行特化，并在里面放进一个 `true` 即可：

```

template<> struct SupportsBitwiseCopy<MyType>
{
    enum { result = true };
};

```

2.10.6 包装

表 2.1 整理出一组由 Loki 定义并实作的完整 **traits**。

2.11 摘要

以下是本书各组件用到的技术。大部分技术都和 `template` 有关。

- 编译期 `assertions` (2.1 节)，帮助程序库为泛型代码产生有意义的错误消息。
- 模板偏特化 (*Partial template specialization*, 2.2 节)，让你可以特化 `template` — 并非针对特定的、固定集合的参数，而是针对“吻合某个式样的一群参数”。
- 局部类 (*Local classes*, 2.3 节)，让你做些有趣的事，特别是对 `template` 函数。
- 常整数映射为型别 (*Mapping integral constants to types*, 2.4 节)，允许在编译期以数值（特别是 `boolean`）作为分派 (`dispatch`) 的取决因素。
- 型别对型别的映射 (*Type-to-type mapping*, 2.5 节)，让你利用函数重载取代 C++ 缺乏的一个特性：函数模板偏特化 (*function template partial specialization*)。
- 型别选择 (*Type selection*, 2.6 节)，让你得以根据 `boolean` 条件来选择型别。
- 编译期间侦测可转换性 (*convertibility*) 和继承性 (2.7 节)，让你得以判断任意两型别是否可互相转换，或是否为相同型别，或是否有继承关系。

表 2.1 `TypeTraits<T>` 的各个成员

名称	种类	说明
<code>isPointer</code>	Boolean 常数	如果 <code>T</code> 是指针, 此值为 <code>true</code>
<code>PointeeType</code>	Type	如果 <code>T</code> 是个指针型别, 此式求得 <code>T</code> 所指型别。如果 <code>T</code> 不是指针型别, 核定结果为 <code>NullType</code>
<code>isReference</code>	Type	如果 <code>T</code> 是个 <code>reference</code> 型别, 核定结果为 <code>true</code>
<code>ReferencedType</code>	Type	如果 <code>T</code> 是个 <code>reference</code> 型别, 此式求得 <code>T</code> 所指型别。否则求得 <code>T</code> 自身型别
<code>ParameterType</code>	Type	此式求得“最适合作为一个 <code>nonmutable</code> 函数 (译注: 不会更改操作对象内容) 的参数”的型别。可以是 <code>T</code> 或 <code>const T&</code>
<code>isConst</code>	Boolean 常数	如果 <code>T</code> 是个常数型别 (经 <code>const</code> 修饰), 则为 <code>true</code>
<code>NonConstType</code>	Type	将型别 <code>T</code> 的 <code>const</code> 饰词拿掉 (如果有的话)
<code>isVolatile</code>	Boolean 常数	如果 <code>T</code> 是个经 <code>volatile</code> 修饰的型别, 则为 <code>true</code>
<code>NonVolatileType</code>	Type	将型别 <code>T</code> 的 <code>volatile</code> 饰词拿掉 (如果有的话)
<code>NonQualifiedType</code>	Type	将型别 <code>T</code> 的 <code>const</code> 和 <code>volatile</code> 饰词拿掉 (如有的话)
<code>isStdUnsignedInt</code>	Boolean 常数	如果 <code>T</code> 是四个不带正负号的整数型别之一 (<code>unsigned char</code> , <code>unsigned short int</code> , <code>unsigned int</code> , <code>unsigned long int</code>), 此值为 <code>true</code>
<code>isStdSignedInt</code>	Boolean 常数	如果 <code>T</code> 是四个带正负号的整数型别之一 (<code>char</code> , <code>short int</code> , <code>int</code> , <code>long int</code>), 此值为 <code>true</code>
<code>isStdIntegral</code>	Boolean 常数	如果 <code>T</code> 是个标准整数型别, 此值为 <code>true</code>
<code>isStdFloat</code>	Boolean 常数	如果 <code>T</code> 是个标准浮点数型别 (<code>float</code> , <code>double</code> , <code>long double</code>), 此值为 <code>true</code>
<code>isStdArith</code>	Boolean 常数	如果 <code>T</code> 是个标准算术型别 (整数或浮点数), 此值为 <code>true</code>
<code>isStdFundamental</code>	Boolean 常数	如果 <code>T</code> 是个基本型别 (算术型别或 <code>void</code>), 此值为 <code>true</code>

- `TypeInfo` (2.8 节) 实作出一个包装了 `std::type_info` 的 `template class`, 其内包含 `value` 语义和次序比较 (`ordering comparisons`) 等特性。
- `NullType` 和 `EmptyType` (2.9 节), 其功能犹如在 `template metaprogramming` 中的占位型别 (`placeholder types`)。
- `TypeTrait` (2.10 节) 提供许多一般用途的 `traits`, 让你可以根据不同的型别裁制你的代码。

3

Typelists

Typelists 是一个用来操作一大群型别的 C++ 工具。就像 lists 对数值提供各种基本操作一样，typelists 对型别也提供相同操作。

有些设计模式 (design patterns) 具体指定并操作一群型别，其中也许有继承关系 (但也许没有)。显著的例子是 Abstract Factory 和 Visitor (Gamma et al. 1995)。如果以传统编程技术来操作一大群型别，将是全然的重复性工作。如此重复会导致隐微的代码膨胀。多数人不会想到其实它可以比现在更好。Typelists 带给你一种能力，可以将经常性的宏工作自动化。Typelists 将来自外星球的强大威力带到 C++ 中，让它得以支持新而有趣的一些手法 (idioms)。

本章介绍一个专为 C++ 设计的 **typelist** 完整设施，以及许多运用范例。阅读本章之后，你将：

- 对 **typelist** 的概念有所了解
- 了解 **typelists** 如何被产生及运作
- 能够更高效地操作 **typelists**
- 了解 **typelists** 的主要用途，以及它们可支持的编程手法 (programming idioms)

第 9、10、11 章都以 **typelists** 作为前提技术。

3.1 Typelists 的必要性

有时候你必须针对某些型别重复撰写相同的代码，而且 **templates** 无法帮上忙。假设你需要实作一个 Abstract Factory (Gamma et al. 1995)。Abstract Factory 规定你必须针对设计期间已知的一群型别中的每一个型别定义一个虚函数，像这样：

```
class WidgetFactory
{
public:
    virtual Window* CreateWindow() = 0;
    virtual Button* CreateButton() = 0;
    virtual ScrollBar* CreateScrollBar() = 0;
};
```

如果你想将 `AbstractFactory` 的概念泛化，并将它纳入程序库中，你必须让使用者得以产生针对任意型别（而不只是 `Window`、`Button` 和 `ScrollBar`）的工厂（`factories`）。`templates` 无法支持这一特性。

虽然一开始 `AbstractFactory` 似乎没有提供太多抽象和泛化的机会，但还是有些事情值得研究：

1. 如果你不试图泛化基础概念，就不太有机会泛化这些概念的具象实体。这是很重要的原则。如果你没能将本质（`essence`）泛化，你仍然得和本质所派生的具象实体纠缠奋战。在 `AbstractFactory` 中，虽然抽象的 `base class` 十分简单，但你会在实作各式各样 `factories` 时遇到很多烦人又重复的代码。
2. 你无法轻易操作 `widgetFactory` 成员函数（见稍早的代码）。本质上我们不可能以泛型方式处理一群虚函数标记式（`virtual function signatures`）。例如，请考虑如下：

```
template <class T>
T* MakeRedWidget(WidgetFactory& factory)
{
    T* pW = factory.CreateT(); // huh???
    pW->SetColor(RED);
    return pW;
}
```

你得根据 `T` 是个 `Window`、`Button` 或 `ScrollBar`，分别调用 `CreateWindow()`、`CreateButton()` 或 `CreateScrollBar()`，但 C++ 不允许你做这样的文字替换。

3. 最后一点（但并非不重要），优秀程序库可摆脱“命名习惯的无尽争议（到底是 `createWindow` 好呢，还是 `create_window` 或 `CreateWindow?`）”。它们引入一个更好、更标准化的方法来做这些事。大致而言，`AbstractFactory` 的确有这些良好副作用。

让我们把以上三点放到需求清单中。为满足第一点，我们最好能够将一串参数传给 `AbstractFactory template`，产出一个 `widgetFactory`，像这样：

```
typedef AbstractFactory<Window, Button, ScrollBar> WidgetFactory;
```

为满足第二点，我们需要对各种 `CreateXxx()` 函数有一个类似 `template` 的调用形式，例如 `Create<Window>()`，`Create<Button>()` 等等，然后就可以在泛型程序中这样调用它们：

```
template <class T>
T* MakeRedWidget(WidgetFactory& factory)
{
    T* pW = factory.Create<T>(); // aha!
    pW->SetColor(RED);
    return pW;
}
```

然而，我们无法满足这样的需求。首先，先前的 `WidgetFactory` 型别定义是不可能的，因为 `templates` 无法拥有不定量参数。其次 `template` 语法 `Create<Xxx>` 不合法，因为虚函数不可以是 `templates`。

通过这些分析，我想你看到了，我们有多好的抽象化和复用化的机会，开发这些机会时我们又将遇到多糟的语言限制。

`Typelists` 将使 `Abstract Factories` 泛化成真，并带来更多其他利益。

3.2 定义 Typelists

由于种种因素，C++ 是一种会让其使用者有时说出“这是我写过的最精巧的五行代码”的程序语言。这或许是由于其语义的丰富性，或源于其总是令人兴奋（并惊讶）的特性。与这种传统一致的是，`typelists` 的本质亦十分简朴：

```
namespace TL
{
    template <class T, class U>
    struct Typelist
    {
        typedef T Head;
        typedef U Tail;
    };
}
```

首先，所有 `typelists` 相关东西都定义于 `TL` 命名空间内，而 `TL` 位于 `Loki` 命名空间中，因此整个都算是 `Loki` 代码。为了简化范例，本章略而未写 `TL` 命名空间。当你使用 `Typelist.h` 时可别疏忽了这一点（如果你忘记了，编译器会提醒你）。

`Typelist` 持有两个型别，我们可通过其内部型别 `Head` 和 `Tail` 加以访问。就这样了！我们不需要持有三个或更多个型别的 `typelist`，因为我们已经有了。例如下面就是有着三个 C++ `char` 型别变体的 `typelist`（注意尾端两个 `>` 符号之间需要空格，这很烦人，但是必要）：

```
typedef Typelist<char, Typelist<signed char, unsigned char> >
CharList;
```

`Typelists` 内部没有任何数值（`value`）：它们的实体是空的，不含任何状态（`state`），也未定义任何函数。执行期间 `typelists` 不带任何数值。它们存在的理由只是为了携带型别信息。因此，对 `typelist` 的任何处理都一定发生在编译期而非执行期。`Typelists` 并未打算被具现化——虽然这样做也没什么损害。因此，本书无论何时提到“`a typelist`”，实际指的是一个 `typelist` 型别，不是一个 `typelist` 对象——那不是我们所关注的。`typelist` 的型别才是有用的（3.13.2 节展示如何以 `typelist` 建立一组对象）。

这里使用的性质是：`template` 参数可以是任何型别，包含该 `template` 的其他具现体。这是一个众所周知的 `template` 特性，尤其常被用来实作 `array`，例如 `vector< vector<double> >`。由于

`Typelist` 接受两个参数，所以我们总是可以藉由“将其中一个参数置换为另一个 `Typelist`”来达到无限延伸的目的。

然而有个小问题。虽然我们可以表达出持有两个型别或更多型别的 `typelists`，但我们无法表达出持有零个或一个型别的 `typelist`。我们需要一个 *null list type*，而第2章的 `NullType` 正适合这样的用途。

现在我来制定一个习惯：每个 `typelist` 都必须以 `NullType` 结尾。`NullType` 可被视为一个结束记号，类似传统 C 字符串的 `\0` 功能。现在我们可以定义一个只持有单一元素的 `typelist` 如下：

```
// See Chapter 2 for the definition of NullType
typedef Typelist<int, NullType> OneTypeOnly;
```

内含三个 `char` 变体的 `typelist` 则是像下面这样：

```
typedef Typelist<char, Typelist<signed char,
    Typelist<unsigned char, NullType> > > AllCharTypes;
```

由此我们得以拥有一个无限的 `Typelist` template，藉由组合“基本单元”而持有任意量的型别。

现在让我们看看如何操作 `typelists`（再一次强调，这里指的是 `Typelist` 型别而非 `Typelist` 对象）。准备好开始有趣的旅程吧。从这儿开始我们将钻研 C++ 地下技术，一个由奇异和新规则组成的世界，一个“编译期编程世界”（*compile-time programming*）。

3.3 将 `Typelist` 的生成线性化（*linearizing*）

详细讨论之前，我必须先说，`typelists` 实在太过偏向 LISP 风格了，以至于不好使用。LISP 风格对 LISP 程序员当然好，对 C++ 程序员却不怎么对味（两个 `'>'` 之间的空格没什么好说的，不过你还是得注意它）。例如以下是一个整数型别的 `typelist`：

```
typedef Typelist<signed char,
    Typelist<short int,
        Typelist<int,
            Typelist<long int, NullType> > > >
    SignedIntegrals;
```

`Typelists` 或许是个酷点子，不过很显然它需要更好的包装。

为了将 `typelist` 的生成线性化，`typelist` 程序库（Loki 中的 `Typelist.h`）定义出很多宏，用来将递归形式转换成比较简单的枚举形式，取代冗长的重复动作。这不是问题，重复工作只需在程序库中做一次即可。Loki 把 `typelists` 的长度扩充到 50（这是 Loki 内部定义的一个数值）。这些宏看起来像这样：

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2)>
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3)>
#define TYPELIST_4(T1, T2, T3, T4) Typelist<T1, TYPELIST_3(T2, T3, T4)>
...
#define TYPELIST_50(...) ...
```

每个宏使用前一个宏，这让程序库使用者可以轻易扩充上限——如果有此需求的话。

现在我们可以把先前的 `SignedIntegrals` 定义式以更优雅的方式表现出来：

```
typedef TYPELIST_4(signed char, short int, int, long int)
SignedIntegrals;
```

将 `typelist` 的生成线性化，只是包装手法的一个开端。`Typelist` 的操作仍然非常笨拙。例如访问 `SignedIntegrals` 最后一个元素竟需用到 `SignedIntegrals::Tail::Tail::Tail`。这对我们“泛化操作 `typelists`”而言还是不够清晰易懂。现在我们应该思考传统的 `lists` 提供哪些基本操作，并以此定义出 `typelist` 的一些基本操作。

3.4 计算长度

这是一个简单的操作。假设有一个 `TList typelist`，它有一个编译期常数代表其长度。这个常数理当是个编译期常数，因为 `typelists` 是一种静态构件，我们预期所有与 `typelists` 相关的计算都将在编译期完成。

大部分 `typelist` 操作函数的基本概念是，开拓“递归式(recursive)templates”，这是指某种 `templates` 以其本身具现体当做其定义的一部分。当这么做时，它们会传递一个不同的 `template` 引数列表。这种形式所产生的递归将止于一个“被作为边界情况运用”的特化体 (`explicit specialization`)。

下面这段用来计算 `typelist` 长度的代码十分简明：

```
template <class TList> struct Length;
template <> struct Length<NullType>
{
    enum { value = 0 };
};
template <class T, class U>
struct Length<Typelist<T, U> >
{
    enum { value = 1 + Length<U>::value };
};
```

以 C++ 的说法是：“`null typelist` 的长度为 0，其他 `typelist` 的长度是 ‘tail 的长度加 1’ ”。

实作 `Length` 时需要运用 `template` 偏特化 (`partial specialization`，参见第 2 章) 来区分 `null type` 和 `typelist`。上述第一版本是 `Length` 全特化，只匹配 `NullType`。第二版本是 `Length` 偏特化，可匹配任何 `Typelist<T,U>`，包括“复合型 (compound) `typelists`”——亦即 `U` 本身又是个 `Typelist<V,W>`。

第二份特化完成了递归式运算。它将 `value` 定义为数值 1（用来将 `head` 计算进去）加上 `tail` 的长度。当 `tail` 变成 `NullType` 时，吻合第一份特化定义，于是停止递归并巧妙地传回结果。这便是长度计算过程。如果你想定义一个 `C array`，用来存放指针（指向所有带正负号整数之 `std::type_info` 对象），有了 `Length` 你便可以这么写：

```
std::type_info* intsRtti[Length<SignedIntegrals>::value];
```

于是，通过编译期的运算，你为 `intsRtti` 分配了 4 个元素⁸。

3.5 间奏曲

你可以在 Veldhuizen(1995)找到一些早期的 `template meta-programs` 实例。Czarnecki 和 Eisenecker (2000) 很深入地讨论过这个问题，并提供一个完整的编译期“C++ 语句组”模拟。

`Length` 的概念和实作很类似计算机科学中的经典递归范例：一个用来计算单向 `linked list` 长度的算法（虽然两者之间其实还有两点主要不同：`Length` 所用的算法在编译期执行，而且它只操作型别，不操作实物）。

这很自然引导出一个问题：我们不能为 `Length` 发展一个迭代（`iteration`）版本用以取代递归（`recursion`）版本吗？毕竟迭代对 C++ 来说比递归更自然。而且为了获得答案，我们可能进而发展出其他 `TypeList` 设施。

答案是否定的，原因很有趣。

我们的“C++ 编译期编程”工具是：`template`、编译期整数计算、`typedefs`。现在让我们看看这些工具以什么方式来帮助我们。

- `Templates`——更明确地说是指 `template specialization`（模板特化）——提供编译期间的 `if` 叙述。一如先前见过的 `Length`，特化版本能够在 `typelists` 和其他型别之间形成差异。
- `Integer calculations`（整数计算）提供真实的数值计算能力，用以从型别转为数值。然而其中有些古怪：所有编译期数值都是不可变的（`immutable`），一旦你为它定义了一个整数常数，例如一个枚举值（`enumerated value`），就不能再改变它（也就是说不能重新赋予他值）。
- `typedefs` 可被视为用来引进“具名的型别常数”（`named type constants`）。它们也是定义之后就被冻结——你不能将 `typedef` 定义的符号重新定义为另一个型别。

编译期计算有两点特点，使它根本上不兼容于迭代（`iteration`）。所谓迭代是持有一个迭代器（`iterator`）并改变它，直到某些条件吻合。由于编译期间我们并没有“可资变化的任何东西”（`mutable entities`），所以无法实现“迭代”。因此，虽然 C++ 是一种极重要的语言，但任何编译期运算所倚赖的技术明显让人联想到过去的纯粹函数型（`functional`）语言——那些语言不会改变数量内容。现在，准备接受高剂量递归吧。

⁸ 你也可以不靠重复的代码完成 `array` 初始化。这留给读者作为练习。

3.6 索引式访问 (Indexed Access)

通过索引来访问 `typelist` 元素，这种能力令人向往。这将使 `typelist` 的访问线性化，使我们能更轻松的操作 `typelist`。当然啦，就像我们在 `static` 世界中操作的所有物体一样，索引必须是个编译期数值 (`compile-time value`)。

一个带有索引操作的 `template` 声明式，看起来像这样：

```
template <class TList, unsigned int index> struct TypeAt;
```

现在让我们来定义算法。记住，不能使用任何可变量值 (`mutable value`, `modifiable value`)。

TypeAt

输入: `typelist TList`, 索引值 `i`

输出: 内部某型别 `Result`

如果 `TList` 不为 `null`, 且 `i` 为 0, 那么 `Result` 就是 `TList` 的头部。

否则

如果 `TList` 不为 `null` 且 `i` 不为 0, 那么 `Result` 就是“将 `TypeAt` 施行于 `TList` 尾端及 `i-1`”的结果。

否则, 逾界 (`out-of-bound`) 访问, 造成编译错误。

下面就是 `TypeAt` 算法的化身：

```
template <class Head, class Tail>
struct TypeAt<Typelist<Head, Tail>, 0>
{
    typedef Head Result;
};
template <class Head, class Tail, unsigned int i>
struct TypeAt<Typelist<Head, Tail>, i>
{
    typedef typename TypeAt<Tail, i-1>::Result Result;
};
```

如果你试着逾界访问，编译器会抱怨找不到 `TypeAt<NullType, x>` 特化版本，其中 `x` 是你所指定的逾界索引。这个消息还可以更好一些，但这样也就不错了。

Loki 的 `Typelist.h` 还定义了一个 `TypeAt` 变型，名为 `TypeAtNonStrict`，实作出类似 `TypeAt` 的功能，不同之处是它对逾界访问更加宽容，以 `NullType` 为回传值，取代编译错误消息。第 5 章介绍的“泛化 `callback` 实作代码”使用上了 `TypeAtNonStrict`。

对 `typelist` 进行索引访问，花费的时间与 `typelist` 大小有关。对 `value list` 来说这种方法不够高效（所以 `std::list` 并未定义 `operator[]`），然而对 `typelist` 来说，时间花在编译期，就某种意义上来说这是“免费的”⁹。

⁹ 事实上，对大型项目而言这种说法不十分正确。至少理论上我们知道，大量 `typelist` 的操作可能导致编译时间拉长。无论如何，程序如果内含大型 `typelists`，那么若非造成执行期对速度十分饥渴（于是你可能只得接受较慢的编译），就是太过耦合 (`coupled`)，于是你可能需要重新检阅你的设计。

3.7 查找 Typelists

如何在 `typelist` 中找到某个型别呢？让我们试着实作出 `IndexOf` 算法，用以计算 `typelist` 内某型别所在位置。如果找不到就传回一个非法值 `-1`。这个算法是古典的线性查找，以递归方式完成。

IndexOf

输入: `typelist TList, type T`

输出: 内部编译期常数 `value`

如果 `TList` 是 `NullType`，令 `value` 为 `-1`。

否则

 如果 `TList` 的头端是 `T`，令 `value` 为 `0`。

 否则

 将 `IndexOf` 施行于“`TList` 尾端和 `T`”，并将结果置于一个暂时变量 `temp`。

 如果 `temp` 为 `-1`，令 `value` 为 `-1`。

 否则令 `value` 为 `1+temp`。

`IndexOf` 是一个相对简单的算法。需特别注意的是如何将“没找到”（`value` 为 `-1`）往外传为计算结果。我们需要三个特化版本，每个版本对应算法中的一个分支点。最后一个分支（根据 `temp` 计算 `value`）是个数值计算，我以条件操作符 `?:` 完成。下面是实作码：

```
template <class TList, class T> struct IndexOf;
template <class T>
struct IndexOf<NullType, T>
{
    enum { value = -1 };
};
template <class Tail, class T>
struct IndexOf<Typelist<T, Tail>, T>
{
    enum { value = 0 };
};
template <class Head, class Tail, class T>
struct IndexOf<Typelist<Head, Tail>, T>
{
private:
    enum { temp = IndexOf<Tail, T>::value };
public:
    enum { value = temp == -1 ? -1 : 1 + temp };
};
```

3.8 附加元素至 Typelists

我们还需要一个可将“某个 type 或整个 typelist”加入 typelist 的工具。先前已讨论过，修改 typelist 是不可能的，但我们将以 *by value* 方式传回一个我们所期望的新 typelist。

Append

输入：typelist TList, type or typelist T

输出：内部某型别 Result

如果 TList 是 NullType 而且 T 是 NullType，那么令 Result 为 NullType。

否则

 如果 TList 是 NullType，且 T 是个 type（而非 typelist），那么 Result 将是“只含唯一元素 T”的一个 typelist。

 否则

 如果 TList 是 NullType，且 T 是一个 typelist，那么 Result 便是 T 本身。

 否则，如果 TList 是 non-null，那么 Result 将是个 typelist，以 TList::Head 为其头端，并以“T 附加至 TList::Tail”的结果为其尾端。

这个算法对应下列代码：

```
template <class TList, class T> struct Append;
template <> struct Append<NullType, NullType>
{
    typedef NullType Result;
};
template <class T> struct Append<NullType, T>
{
    typedef TYPELIST_1(T) Result;
};
template <class Head, class Tail>
struct Append<NullType, Typelist<Head, Tail> >
{
    typedef Typelist<Head, Tail> Result;
};
template <class Head, class Tail, class T>
struct Append<Typelist<Head, Tail>, T>
{
    typedef Typelist<Head,
        typename Append<Tail, T>::Result>
        Result;
};
```

再次请注意最后一个 Append 偏特化版本，它递归具现 Append，每次递归都将 tail 和“待附加型别”传递进去。

现在，面对单一 `type` 和 `typelist` 两者，我们有了一致的 `Append` 操作型式。下面这个语句：

```
typedef Append<SignedIntegrals,
    TYPELIST_3(float, double, long double)>::Result
    SignedTypes;
```

便是定义一个 `typelist`，涵盖 C++ 所有带正负号 (signed) 数值型别。

3.9 移除 Typelist 中的某个元素

现在考虑“附加”的相对操作：从一个 `typelist` 中“移除”某个型别。我们有两个选择：只移除第一个出现个体，或是移除所有出现个体。

首先让我们考虑只移除第一个出现者。

Erase

输入： `typelist TList, type T`

输出：内部某型别 `Result`

如果 `TList` 是 `NullType`，那么 `Result` 就是 `NullType`。

否则

 如果 `T` 等同于 `TList::Head`，那么 `Result` 就是 `TList::Tail`。

 否则，`Result` 将是一个 `typelist`，它以 `TList::Head` 为头端，并以“将 `Erase` 施行于 `TList::Tail` 和 `T`”所得结果为尾端。

以下是上述算法对应的 C++ 代码：

```
template <class TList, class T> struct Erase;
template <class T> // Specialization 1
struct Erase<NullType, T>
{
    typedef NullType Result;
};
template <class T, class Tail> // Specialization 2
struct Erase<Typelist<T, Tail>, T>
{
    typedef Tail Result;
};
template <class Head, class Tail, class T> // Specialization 3
struct Erase<Typelist<Head, Tail>, T>
{
    typedef Typelist<Head,
        typename Erase<Tail, T>::Result>
        Result;
};
```

如同 `TypeAt` 一样，这里也没有 `template` 缺省版本，这意味着你不能以任意型别来具现化 `Erase`。例如 `Erase<double, int>` 会导致编译错误，因为它没有匹配者。`Erase` 的第一参数必须是个 `typelist`。

继续沿用先前的 `SignedTypes` 定义，现在我们可以写出这样的代码：

```
// SomeSignedTypes contains the equivalent of
// TYPELIST_6(signed char, short int, int, long int,
// double, long double)
typedef Erase<SignedTypes, float>::Result SomeSignedTypes;
```

另一项移除操作是 `EraseAll`，它会移除 `typelist` 中某个型别的所有出现个体。其实作手法类似 `Erase`，唯一不同的是，发现移除对象后算法并不停止下来，而是继续查找下一个符合条件的元素并删除之，直到 `list` 尾端。

```
template <class TList, class T> struct EraseAll;
template <class T>
struct EraseAll<NullType, T>
{
    typedef NullType Result;
};
template <class T, class Tail>
struct EraseAll<Typelist<T, Tail>, T>
{
    // Go all the way down the list removing the type
    typedef typename EraseAll<Tail, T>::Result Result;
};
template <class Head, class Tail, class T>
struct EraseAll<Typelist<Head, Tail>, T>
{
    // Go all the way down the list removing the type
    typedef Typelist<Head,
        typename EraseAll<Tail, T>::Result>
        Result;
};
```

3.10 移除重复元素 (Erasing Duplicates)

`typelist` 的另一项重要操作就是移除重复元素。第 11 章的 `static double-dispatch engine` 广泛运用了此项机能。

这儿的需求是：转换 `typelist`，让每种型别只出现一次。例如面对下面这个 `typelist`：

```
TYPELIST_6(Widget, Button, Widget, TextField, ScrollBar, Button)
```

我们希望获得这样的 `typelist`：

```
TYPELIST_4(Widget, Button, TextField, ScrollBar)
```

过程有点复杂，但也许你猜到了，我们可以通过 `Erase` 获得一些帮助。

NoDuplicates

输入：typelist TList

输出：内部某型别 Result

如果 `TList` 是 `NullType`, 那么就令 `Result` 为 `NullType`。

否则

将 `NoDuplicates` 施行于 `TList::Tail` 身上, 获得一个暂时的 `typelist L1`。

将 `Erase` 施行于 `L1` 和 `TList::Head`。获得结果 `L2`。

`Result` 是个 `typelist`, 其头端为 `TList::Head`, 尾端为 `L2`。

现在把上述算法转换为代码:

```
template <class TList> struct NoDuplicates;
template <> struct NoDuplicates<NullType>
{
    typedef NullType Result;
};
template <class Head, class Tail>
struct NoDuplicates< Typelist<Head, Tail> >
{
private:
    typedef typename NoDuplicates<Tail>::Result L1;
    typedef typename Erase<L1, Head>::Result L2;
public:
    typedef Typelist<Head, L2> Result;
};
```

为什么当我们以为 `EraseAll` 较为恰当的时候, `Erase` 就够用了呢——我们不是希望移除所有重复型别吗? 答案是, `Erase` 被实施于 `NoDuplicates` 递归操作之后。这意味着我们将从 `list` 之中移除的是一个“不再有任何重复个体”的型别, 所以最多只会会有一个型别个体 (instance of the type) 被移除。这一段递归编程相当有趣。

3.11 取代 `Typelist` 中的某个元素

有时候我们需要元素的“代换”而非“移除”。一如你将于 3.12 节所见, 将某个型别取代为另一个型别是惯用技法 (idioms) 中很重要的部分。

我们需要在一个 `typelist TList` 中以型别 `U` 取代型别 `T`。

Replace

输入: `typelist TList`, `type T` (被取代者), 以及 `type U` (取代者)

输出: 内部某型别 `Result`

如果 `TList` 是 `NullType`, 令 `Result` 为 `NullType`。

否则

如果 `typelist TList` 的头端是 `T`, 那么 `Result` 将是一个 `typelist`, 以 `U` 为其头端并以 `TList::Tail` 为其尾端。

否则 `Result` 是一个 `typelist`, 以 `TList::Head` 为其头端, 并以“`Replace` 施行于 `TList`, `T`, `U`”的结果为其尾端。

一旦理解上述递归算法, 很自然可写出以下这样的代码:

```

template <class TList, class T, class U> struct Replace;
template <class T, class U>
struct Replace<NullType, T, U>
{
    typedef NullType Result;
};
template <class T, class Tail, class U>
struct Replace<Typelist<T, Tail>, T, U>
{
    typedef Typelist<U, Tail> Result;
};
template <class Head, class Tail, class T, class U>
struct Replace<Typelist<Head, Tail>, T, U>
{
    typedef Typelist<Head,
        typename Replace<Tail, T, U>::Result>
        Result;
};

```

如果改变上述第二个特化版本, 将算法递归施行于 `Tail` 身上, 轻易便可获得 `ReplaceAll` 算法 (译注: 请看 Loki "typelist.h" 便知道实际做法)。

3.12 为 Typelists 局部更换次序 (Partially Ordering)

假设我们打算根据继承关系进行排序, 例如希望派生型别出现在基础型别之前。举个例子, 我们手上有一个图 3.1 所示的 class 继承体系:

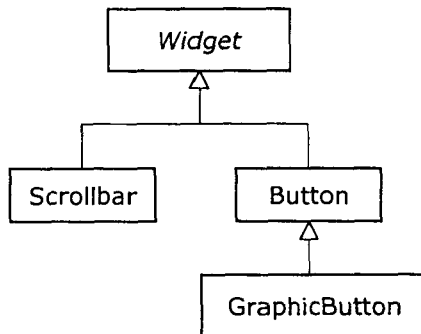


图 3.1 一个简单的 class 继承体系

如果我们有这样一个 typelist:

```
TYPELIST_4(Widget, ScrollBar, Button, GraphicButton)
```

现在的挑战是如何将它转换为:

```
TYPELIST_4(ScrollBar, GraphicButton, Button, Widget)
```

也就是把“最末端派生型别”(most derived types)带向前,并让其他兄弟型别的顺序不变。

这似乎只是个智力练习,但其实有其重要应用。重新排列一个 typelist 的元素次序,使最末端派生型别放在最前面,可以确保由下而上走访(遍历)这个 class 继承体系。第11章的 double-dispatch engine 便是运用这个重要技术来解析型别信息。

当我们准备对一群型别排序时,需要一个排序函数。先前已经有了一个在编译期用来侦测继承关系的工具,第2章曾有详细讨论。回想一下,我们有一个方便的宏 SUPERSUBCLASS(T,U),如果 U 派生自 T,它会传回 true。太好了,我们只需将“继承侦测机制”合并到 typelist 即可。

这里不能使用充分而成熟的排序算法,因为我们没有完整的顺序关系:是的,对 classes 而言并没有与 operator< 相对应的东西,于是本例两个兄弟型别无法被 SUPERSUBCLASS(T,U)排序。我将使用一个自制算法,将派生型别带到前面,并令其他 classes 的相对位置保持不变。

DerivedToFront

输入: typelist TList

输出: 内部某型别 Result

如果 TList 是 NullType, 令 Result 为 NullType。

否则

从 TList::Head 到 TList::Tail, 找出最末端派生型别, 存储于暂时变量 TheMostDerived 中。

以 TList::Head 取代 TList::Tail 中的 TheMostDerived, 获得 L。

建立一个 typelist, 以 TheMostDerived 为其头端, 以 L 为其尾端。

将这个算法施行于 typelist 身上, 派生型别便会被移至 typelist 头端, 基础型别被推移至尾端。

这里还缺少一样东西: 在某个 typelist 中查找“某型别之最深层派生型别”的算法。由于 SUPERSUBCLASS 会传回一个编译期 Boolean 值, 我们发现第2章中的一个小型的 select class template 可以派上用场。还记得吗, select 可根据编译期 Boolean 值在两个型别中选择一个。

MostDerived 算法接受一个 **typelist** 和一个 **Base** 型别, 传回 **typelist** 中 **Base** 的最深层派生型别 (如果找不到任何派生型别, 就传回 **Base** 自己)。看来像这样:

MostDerived

输入: **typelist TList**, **type T**

输出: 内部某型别 **Result**

如果 **TList** 是 **NullType**, 令 **Result** 为 **T**。

否则

将 **MostDerived** 施行于 **TList::Tail** 和 **T** 身上, 获得一个 **Candidate**。

如果 **TList::Head** 派生自 **Candidate**, 令 **Result** 为 **TList::Head**。

否则, 令 **Result** 为 **Candidate**。

MostDerived 实作如下:

```
template <class TList, class T> struct MostDerived;

template <class T>
struct MostDerived<NullType, T>
{
    typedef T Result;
};

template <class Head, class Tail, class T>
struct MostDerived<Typelist<Head, Tail>, T>
{
private:
    typedef typename MostDerived<Tail, T>::Result Candidate;
public:
    typedef typename Select<
        SUPERSUBCLASS (Candidate, Head),
        Head, Candidate>::Result Result;
};
```

前述的 **DerivedToFront** 算法便是以 **MostDerived** 为基础, 下面是其实作:

```
template <class T> struct DerivedToFront;

template <>
struct DerivedToFront<NullType>
{
    typedef NullType Result;
};

template <class Head, class Tail>
struct DerivedToFront<Typelist<Head, Tail> >
{
private:
    typedef typename MostDerived<Tail, Head>::Result
```

```

    TheMostDerived;
    typedef typename Replace<Tail,
        TheMostDerived, Head>::Result L;
public:
    typedef Typelist<TheMostDerived, L> Result;
};

```

这个复杂的 `typelist` 操作用处很大。“`DerivedToFront` 转换”可以高效地将“型别处理工序”自动化。没有了它，我们就只能仰赖大量训练和注意力的集中才能完成任务。

3.13 运用 Typelists 自动产生 Classes

如果此刻你认为 `typelist` 很有魅力，很有趣，或是很丑陋，你其实还没真正看到什么东西。我将在这一小节中定义数个基本构想，以 `typelists` 作为代码生成机制。这么一来我们就不必手写太多代码，而是驱动编译器帮我们自动产生代码。这些概念采用了 C++ 最具威力的特性之一，一个不存在于任何其他语言的特性——`template template parameters`。

截至目前，`typelist` 的操作都没有产生真正的代码；运作过程只产生 `typelists`、`types` 或编译期常数（例如 `Length`）。让我们尽可能产生一些真正的代码，也就是真正能够在编译结果中留下足迹的东西。

`Typelist` 对象本身没什么用；它们缺乏执行期状态（`state`）和机能（`functionality`）。从 `typelist` 中产生 `classes` 是很重要的编程需求。应用程序编写者有时需要以一种“`typelist` 所指示的方向”来编写 `classes`——填以虚函数、数据声明或函数实作。我将试着运用 `typelist` 将这样的过程自动化。

由于 C++ 缺乏“编译期迭代或递归宏”（`compile-time iteration or recursive macros`），想为 `typelist` 内含的每个型别加入一些代码是很困难的。你可以运用“`template` 偏特化”手法，组合前述算法，但是在客端实作这样的方案，不但笨拙而且复杂。这时候 `Loki` 可以派上用场。

3.13.1 产生“散乱的继承体系（`scattered hierarchies`）”

`Loki` 提供一个极具威力的工具，可轻易将 `typelist` 里的每一个型别套用于一个由用户提供的基本 `template` 身上。这么一来，将 `typelist` 内的型别分发（`distributing`）至客端代码的一些不便过程就被完全封装于 `Loki` 程序库中，使用者只需自行定义一个单一参数的 `template` 即可。

这样一个 `class template` 名为 `GenScatterHierarchy`。虽然其定义十分简单，很快你会看见 `GenScatterHierarchy` 引擎盖下的惊人马力。下面是其定义式¹⁰：

```

template <class TList, template <class> class Unit>
class GenScatterHierarchy;

```

¹⁰ 此处的情况是，在潜在应用尚未浮现之前，先将设计构想呈现出来。

```

// GenScatterHierarchy specialization: Typelist to Unit
template <class T1, class T2, template <class> class Unit>
class GenScatterHierarchy<TYPELIST_2(T1, T2), Unit>
    : public GenScatterHierarchy<T1, Unit>
    , public GenScatterHierarchy<T2, Unit>
{
};
// Pass an atomic type (nontypelist) to Unit
template <class AtomicType, template <class> class Unit>
class GenScatterHierarchy : public Unit<AtomicType>
{
};
// Do nothing for NullType
template <template <class> class Unit>
class GenScatterHierarchy<NullType, Unit>
{
};

```

template template parameters（见第 1 章）如你预期般地进行大量工作。你传入一个名为 `Unit` 的 `template class` 当做 `GenScatterHierarchy` 的第二参数。`GenScatterHierarchy` 内部对该 `Unit` 的使用就像面对任何“带有单一 `template` 参数”的一般 `template class` 一样。它所能显现的威力来自于你（`GenScatterHierarchy` 使用者）：给它一个你自己写的 `template` 吧。

`GenScatterHierarchy` 做了什么？如果其第一引数是个 **atomic type**（意指单一型别，相对于 `typelist`），`GenScatterHierarchy` 便把该型别传给 `Unit`，然后继承 `Unit<T>`。如果 `GenScatterHierarchy` 的第一引数是个 `typelist TList`，就递归产生 `GenScatterHierarchy<TList::Head, Unit>` 和 `GenScatterHierarchy<TList::Tail, Unit>` 并继承此二者。`GenScatterHierarchy<NullType, Unit>` 则是个空类。

最终，一个 `GenScatterHierarchy` 具现体会继承 `Unit` “对 `typelist` 中每一个型别”的具现体。例如下面这段代码：

```

template <class T>
struct Holder
{
    T value_;
};
typedef GenScatterHierarchy<
    TYPELIST_3(int, string, Widget),
    Holder>
WidgetInfo;

```

由 `WidgetInfo` 产生的继承体系如图 3.2。我们称此种 `class` 继承体系是散乱的（*scattered*），因为 `typelist` 内的型别散乱分布于不同的 `root class` 之下。`GenScatterHierarchy` 的要旨是：它藉由重复具现化一个你所提供的 `class template`（视之为模型），为你产生一个 `class` 继承体系，然后将所有这样产生出来的 `classes` 聚焦至一个 `leaf class`（叶类、末端类），亦即本例的 `WidgetInfo`。

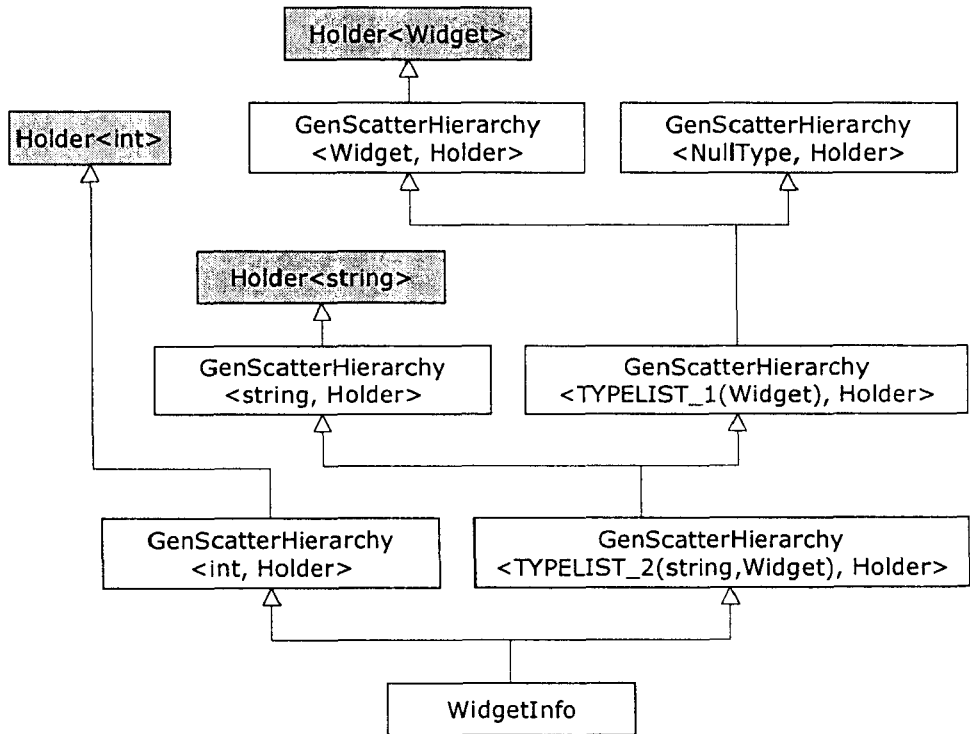


图 3.2 widgetInfo 的继承结构

由于继承了 `Holder<int>`、`Holder<string>` 和 `Holder<widget>`，`widgetInfo` 因而针对它们（亦即 `typelist` 中的每个型别）各自拥有一个成员变量 `value_`。图 3.3 显示 `widgetInfo` 对象的内存分配可能形式。这样的分配形式首先假设空类 `GenScatterHierarchy<NullType, Holder>` 会被优化掉，不会在这个复合对象中占据实体位置。

你还可以对 `widgetInfo` 对象做些有趣的事情。例如以这种方式访问其中的 `string` 对象：

```
WidgetInfo obj;
string name = (static_cast<Holder<string>&>(obj)).value_;
```

其中的显式转型（explicit cast）是必要的，以消除成员变量名称 `value_` 的可能含糊意义，否则编译器不知道你打算取用哪一个 `value_`。

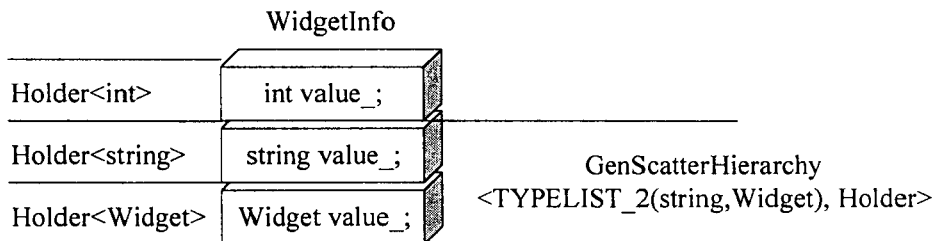


图 3.3 WidgetInfo 的内存分配图

如此的转型十分难看，让我们试着藉由提供某些方便的访问函数，把 `GenScatterHierarchy` 变得更容易使用。例如提供一个成员函数，根据成员的类型来取用成员变量。这很容易：

```
// Access a base class by type name
template <class T, class TList, template <class> class Unit>
Unit<T>& Field(GenScatterHierarchy<TList, Unit>& obj)
{
    return obj;
}
```

`Field` 倚赖 **derived-to-base** 隐式转换。如果你调用 `Field<Widget>(obj)`（其中 `obj` 是一种 `WidgetInfo`），编译器会知道 `Holder<Widget>` 是 `WidgetInfo` 的 base class 并只传回复合对象中该成分的 reference。

为何 `Field` 是一个 namespace-level 函数而不是一个成员函数呢？因为在如此高阶的泛型程序设计中，我们必须小心处理名称。举个例子，想象一下，如果 `Unit` 自己定义了一个名为 `Field` 的符号，而 `GenScatterHierarchy` 很糟糕地自己也定义了一个名为 `Field` 的成员函数，后者将会遮蔽前者。这会成为惹恼使用者的一个主要原因。

`Field` 另有一个问题也是惹恼使用者的主因：当你的 `typelist` 内含重复型别时，你不能使用 `Field`。看一下这个稍作修改的 `WidgetInfo`：

```
typedef GenScatterHierarchy<
    TYPELIST_4(int, int, string, Widget),
    Holder>
WidgetInfo;
```

现在 `WidgetInfo` 有了两个“型别为 `int`”的 `value_` 成员。如果你试着对某个 `WidgetInfo` 对象调用 `Field<int>`，编译器会抱怨出现模棱两可（歧义）情况。这个问题无法轻松解决，因为 `WidgetInfo` 最终确实通过了不同的路径继承 `Holder<int>` 两次，如图 3.4 所示。

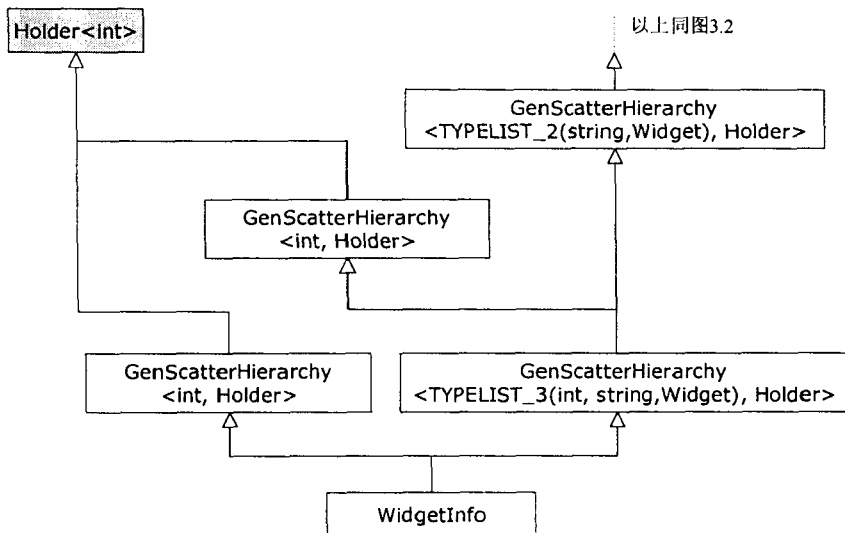


图 3.4 widgetInfo 继承 Holder<int> 两次

因此，我们需要一个“以索引选择 GenScatterHierarchy 实体栏位（fields）”的方法，而非通过型别名称。如果你可以藉由在 typelist 中的位置指出两个 int 栏位（例如这么写：Field<0>(obj) 和 Field<1>(obj)），就可以摆脱模棱两可（歧义）的困境。

让我们试着实作一个基于索引的栏位访问函数。我们必须在编译期分派（dispatch）索引值零（用以访问 typelist head）和非零（用以访问 typelist tail）。有了第 2 章定义的 Int2Type template 的协助，要做分派动作（dispatching）是很容易的。Int2Type 可以直接将不同常数转换为不同型别。

```

template <class TList, template <class> class Unit>
Unit<TList::Head>& FieldHelper(
    GenScatterHierarchy<TList, Unit>& obj,
    Int2Type<0>)
{
    GenScatterHierarchy<TList::Head, Unit>& leftBase = obj;
    return leftBase;
}
template <int i, class TList, template <class> class Unit>
Unit<TypeAt<TList, index>::Result>&
FieldHelper(
    GenScatterHierarchy<TList, Unit>& obj,
    Int2Type<i>)
{
    GenScatterHierarchy<TList::Tail, Unit>& rightBase = obj;
    return FieldHelper(rightBase, Int2Type<i-1>());
}
template <int i, class TList, template <class> class Unit>
Unit<TypeAt<TList, index>::Result>&
Field(GenScatterHierarchy<TList, Unit>& obj)
{
    return FieldHelper(obj, Int2Type<i>());
}

```

写出这样的实作码需要相当的时间，幸运的是它很容易说明。名为 `FieldHelper` 的两个重载函数做了实际工作。第一版本接受一个型别为 `Int2Type<0>` 的参数，第二版本接受的型别是 `Int2Type<any integer>`。因此，第一版本回传的对象相当于 `Unit<T1>&`，第二版本传回的是 `typelist` 之中被索引标示出来的型别。`Field` 和 `FieldHelper` 都用上了 3.6 节定义的 `TypeAt` 算法。`FieldHelper` 第二版本递归调用自己的一个特化体，传入 `GenScatterHierarchy` 的右侧 `base class` 和 `Int2Type<index-1>`。这么做是因为，对任何非零值 `N` 而言，`typelist` 内第 `N` 个栏位其实就是 `tail` 的第 `N-1` 个栏位（`N=0` 的情况则由第一个重载版本负责处理）

为了获得更有效率的接口，我们还需两个额外的 `Field` 函数：两个 `const Field` 函数。它们和 `non-const` 版本类似，但可接受和回传 `references to const types`。

`Field` 使得 `GenScatterHierarchy` 非常容易被使用。现在我们可以这样写：

```

WidgetInfo obj;
...
int x = Field<0>(obj).value_;    // first int
int y = Field<1>(obj).value_;    // second int

```

`GenScatterHierarchy` 很适合从 `typelist` 中产生繁复的 `classes`（只需与一个简单的 `template` 合作）。你可以运用 `GenScatterHierarchy` 对 `typelist` 中的每一个 `types` 产生一些虚函数。第 9 章的 `Abstract Factory` 便运用 `GenScatterHierarchy` 从 `typelist` 产生出抽象生成函数（`abstract creation functions`），该章也展示如何利用 `GenScatterHierarchy` 产生 `classes` 继承体系。

3.13.2 产生 Tuples

有时候你也许只是希望产生一个带有无名栏位的小型结构（某些语言，例如 ML，将这种东西称为一个 **tuple**）。C++ tuple 设施由 Jakko Järvi（1999a）首先提出，而后经过 Järvi 和 Powell（1999b）改良。

什么是 tuples 呢？看看下面的例子：

```
template <class T>
struct Holder
{
    T value_;
};
typedef GenScatterHierarchy<
    TYPELIST_3(int, int, int),
    Holder>
    Point3D;
```

Point3D 的运用方式有点笨拙，因为你必须在每一个栏位访问函数之后写出 `.value_`（译注：就像 p69 下方代码那样）。你需要的其实只是采用和 `GenScatterHierarchy` 相同的方法产生一个结构，但是让 `Field` 访问函数直接传回 `value_ reference`。也就是说 `Field<n>` 不该传回 `Holder<int>&`，应该传回 `int&`。

Loki 定义了一个 `Tuple` template class，其实作手法类似于 `GenScatterHierarchy`，但提供“栏位直接访问”机能。`Tuple` 用起来像这样：

```
typedef Tuple<TYPELIST_3(int, int, int)>
    Point3D;
Point3D pt;
Field<0>(pt) = 0;
Field<1>(pt) = 100;
Field<2>(pt) = 300;
```

Tuples 很适合用来产生一个无任何成员函数的无名结构。例如你可以在函数中利用 tuples 传回多个数值，像这样：

```
Tuple<TYPELIST_3(int, int, int)>
GetWindowPlacement(Window&);
```

上述函数 `GetWindowPlacement` 可让使用者只以一个函数调用就取得一个 `window` 坐标和它在 `windows stack` 中的位置。程序库作者不需要为三个整数型别的 tuples 提供各自不同的结构。

你可以在 `Tuple.h` 中看到 Loki 提供的其他 tuple 相关函数。

3.13.3 产生线性继承体系

考虑下面这个简单的 `template`，它定义了一个事件处理接口（event handler interface），其中只定义了一个成员函数 `OnEvent`：


```

template <class T>
class EventHandler
{
public:
    virtual void OnEvent(const T&, int eventId) = 0;
    virtual void ~EventHandler() {}
};

```

为了策略正确性，EventHandler 还定义了一个虚析构函数，这虽然和我所要讨论的主题无关，却是绝对必要的（原因见第 4 章）。

我们可以运用 GenScatterHierarchy 传发（*distribute*）给 typelist 里任何型别的一个 EventHandler：

```

typedef GenScatterHierarchy
<
    TYPELIST_3(Window, Button, ScrollBar),
    EventHandler
>
WidgetEventHandler;

```

GenScatterHierarchy 的缺点是它使用了多重继承。如果你很在乎对象大小的优化，那么 GenScatterHierarchy 也许就不那么好了，因为 widgetEventHandler 内有三个指向虚函数表（vtables）的指针¹¹，一个指针针对一个 EventHandler 函数实体（译注：此类基础知识可参考 *Inside The C++ Object Model*，《深度探索 C++ 对象模型》）。如果 sizeof(EventHandler) 是 4 bytes，sizeof(WidgetEventHandler) 可能高达 12 bytes，而且会随着你加入 typelist 的型别个数而增加大小。如果希望获得最佳空间使用率，应该把所有虚函数声明到 widgetEventHandler 里头，但这会破坏代码产生机会。

一个好的配置（configuration）是将 widgetEventHandler 分解成“每个虚函数配一个 class”，如图 3.5，是谓“线性继承体系”。藉由单一继承（而非多重继承），widgetEventHandler 只有一个 vtable 指针，达到最佳空间效能。

什么机制才能自动完成这样一个线性继承体系呢？类似 GenScatterHierarchy 的递归性 template 将可带来帮助。然而其中有个不同点：使用者提供的 class template 如今必须接受两个 template 参数，其一是 typelist 内的当前型别（这一点和 GenScatterHierarchy 相同），另一是具现体的 base class。后者之所以需要，因为如图 3.5 所示，客端代码如今需要参与继承体系，而不只是作为根类（像 GenScatterHierarchy 那样）。

现在让我们写出递归的 GenLinearHierarchy template。它很类似 GenScatterHierarchy，不同的是对于“继承关系”和“使用者提供之 template unit”的处理。

¹¹ 并无任何规定说 C++ 编译器一定需要一个虚函数表（virtual tables），不过大部分编译器的确是需要的。关于虚函数表可参考 Lippman(1994)（译注：按该书版权页显示，乃 1996 年出版）

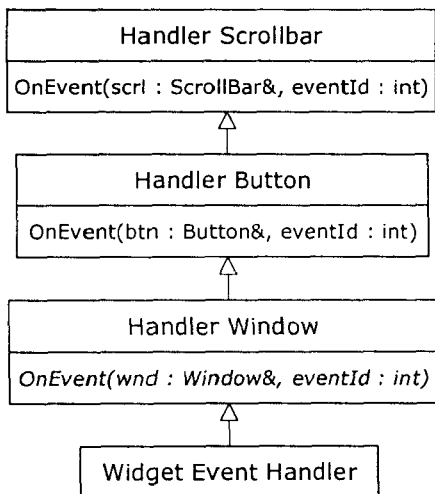


图 3.5 一个有结构大小优化的 WidgetEventHandler

```

template
<
    class TList,
    template <class AtomicType, class Base> class Unit,
    class Root = EmptyType // For EmptyType, consult Chapter 2
>
class GenLinearHierarchy;
template
<
    class T1,
    class T2,
    template <class, class> class Unit,
    class Root
>
class GenLinearHierarchy<Typelist<T1, T2>, Unit, Root>
    : public Unit< T1, GenLinearHierarchy<T2, Unit, Root> >
{ };
template
<
    class T,
    template <class, class> class Unit,
    class Root
>
class GenLinearHierarchy<TYPELIST_1(T), Unit, Root>
    : public Unit<T, Root>
{ };
  
```

这一段代码比 `GenScatterHierarchy` 稍微复杂些，但产出之继承体系结构比较简单。现在让我们验证“一张图胜过千言万语”的箴言，看看图 3.6，其中显示由下列代码产生出来的继承体系。

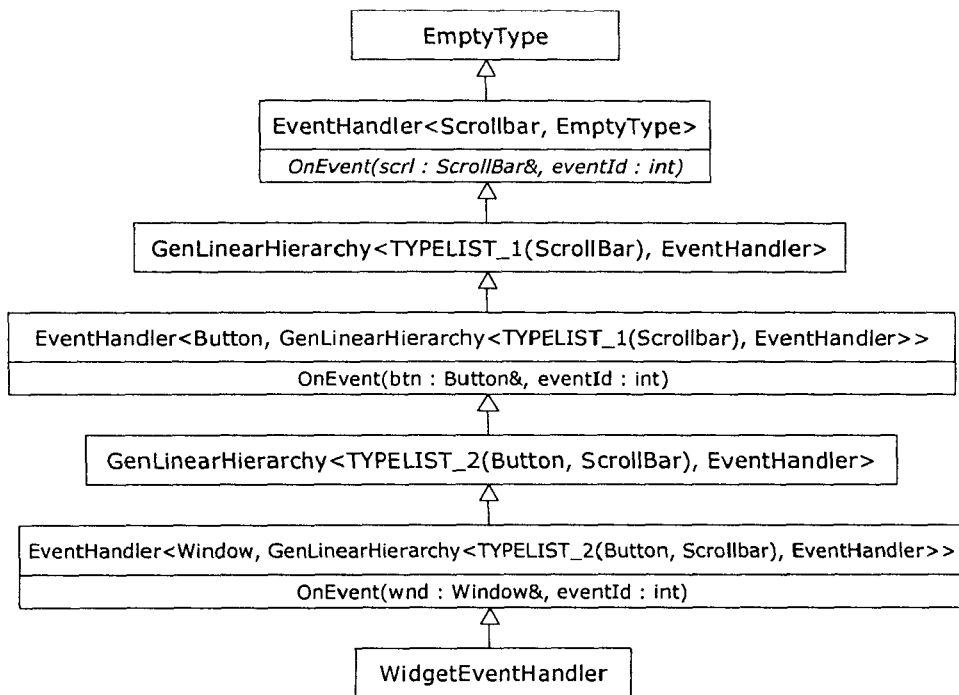


图 3.6 `GenLinearHierarchy` 产生的 class 继承体系

```

template <class T, class Base>
class EventHandler : public Base
{
public:
    virtual void OnEvent(T& obj, int eventId);
};
  
```

```
typedef GenLinearHierarchy
<
    TYPELIST_3(Window, Button, ScrollBar),
    EventHandler
>
MyEventHandler;
```

为了结合 `EventHandler`, `GenLinearHierarchy` 定义了一个线性的、绳索般的、单一继承式的 `classes` 继承体系, 其中任一节点都有一个纯虚函数, 形式如 `EventHandler` 所描述。结果, `MyEventHandler` 一如所求地定义了三个虚函数。`GenLinearHierarchy` 为其 **template parameter** 加上一个新条件: `Unit` (本例中为 `EventHandler`) 必须接受第二个 **template** 引数, 然后成为 `GenLinearHierarchy` 的基类。为了弥补, `GenLinearHierarchy` 十分勤勉地产生 `classes` 继承体系。

要让 `Field` 重载函数提供某些机能 (类似给 `GenScatterHierarchy` 运用的那些) 给 `GenLinearHierarchy` 使用, 是很容易的。`Loki` 总共定义了 8 个 `Field` 重载函数, 分别针对: `GenScatterHierarchy` 和 `GenLinearHierarchy`、`const` 和 `non-const` 对象、“以索引为据”和“以型别为据”。`GenScatterHierarchy` 和 `GenLinearHierarchy` 通常焦不离孟, 大部分情况下你需要以 `GenScatterHierarchy` 产生一个接口, 并以 `GenLinearHierarchy` 实作之。第 9 章和第 10 章实际展示了这两个“class 继承体系产生器”的使用方式。

3.14 摘要

`Typelists` 是一个重要的泛型编程技术。它可以提供程序库作者一些新性能: 表达及操作任意数量的型别, 并从这些型别中产出数据结构和代码, 等等。

编译期间 `typelist` 提供了如同一般 `list` 的绝大部分基本功能: 附加、删除、查找、取用、取代、删除重复, 乃至“依据继承关系局部排序”。操作 `typelist` 的代码, 限制为纯函数风格 (*pure functional style*), 因为缺乏编译期变量可用——不论型别或编译期常数, 一旦被定义, 就再不能被改变了。因为这个缘故, 大部分 `typelist` 操作都倚赖递归式 (*recursive*) `templates`, 并根据 `template` 偏特化 (*partial specialization*) 来完成模式匹配 (*pattern matching*)。

当你必须针对一大群型别撰写相同的代码时, `typelist` 很有用——不论是用于声明或用于实作。它们让你得以抽象化和泛化那些被其他所有泛型编程技术遗忘的东西。正因如此, `typelists` 带来了新特性、新手法、新的程序库实作技术, 一如你将于第 9 章和第 10 章所见。

`Loki` 提供两个强而有力的基本工具 `GenScatterHierarchy` 和 `GenLinearHierarchy`, 让你根据 `typelist` 自动产生继承体系。它们会产生两种 `classes` 结构: 散乱的 (*scatter*, 图 3.2) 和线性的 (*linear*, 图 3.6)。线性继承体系比较高效, 散乱继承体系则具备一个有用性质: 使用者自定义之 `template` 所具现出来的所有实体, 都是最后产出之 `class` 的父类, 如图 3.2。

3.15 Typelist 要点概览

- 头文件: Typelist.h。
- 所有 typelist 工具都归属于 `Loki::TL` 这一命名空间 (namespace)。
- 定义了 `class template Typelist<Head,Tail>`。
- Typelist 的生成: 定义了 `TYPELIST_1` 至 `TYPELIST_50` 共 50 个宏, 它们接受的参数个数如其名称所示。
- 宏上限 (50) 可以扩充如下:

```
#define TYPELIST_51(T1, repeat here up to T51) \
    Typelist<T1, TYPELIST_50(T2, repeat here up to T51) >
```

- 根据习惯, typelists 应该合度——它们总是有个单一型别 (non-typelist) 的头端 (head, 第一元素), 至于尾端 (tail) 可以是 `typelist` 或是个 `NullType`。
- 头文件内定义了一组用来操作 typelists 的基本工具。习惯上, 所有基本工具都传回 `Result`, 那是一个嵌套的 (内部的) `public` 型别。如果操作完毕后传回一个数值, 该数值通常命名为 `value`。
- 上述所谓基本工具, 完整列于表 3.1。

- `class template GenScatterHierarchy` 概要如下:

```
template <class TList, template <class> class Unit>
class GenScatterHierarchy;
```

- `GenScatterHierarchy` 会产生一个继承体系, 由“根据 typelist `TList` 内的每一个型别, 将 `Unit` 具现化”后的所有成果构成。`GenScatterHierarchy` 具现体直接或间接继承自每一个 `Unit<T>`, 其中 `T` 是 typelist 内的任一型别。
- 图 3.2 描绘出 `GenScatterHierarchy` 所产生的继承体系结构。

- `class template GenLinearHierarchy` 概要如下:

```
template <class TList, template <class, class> class Unit>
class GenLinearHierarchy;
```

- 图 3.6 描绘出 `GenLinearHierarchy` 所产生的继承体系结构。
- `GenLinearHierarchy` 会将 typelist `TList` 内的每一个型别传入 `Unit` 当做其第一参数, 以此具现化 `Unit`。很重要的一点: `Unit` 必须以 `public` 方式继承其第二 `template` 参数。
- 重载函数 `Field` 允许使用者“根据型别名称”和“根据索引”访问继承体系中的任一节点。`Loki` 共提供八个 `Field` 重载函数, 包含 `const` 和 `non-const` 版本、“根据型别”和“根据索引”版本, 以及“针对 `GenScatterHierarchy`”和“针对 `GenLinearHierarchy`”版本。
- `Field<Type>(obj)` 传回一个 `reference` 指向某 `Unit` 具现体, 后者相应于特定型别 `Type`。
- `Field<index>(obj)` 传回一个 `reference` 指向某 `Unit` 具现体, 后者相应于“以常数 `index` 标记出来的那个型别”。

表 3.1 作用于 Typelists 身上的各种编译期算法

基本工具 名称	说明
<code>Length<TList></code>	计算 <code>TList</code> 的长度
<code>TypeAt<TList,idx></code>	传回 <code>TList</code> 某位置（以零为基准）上的型别。如果 <code>index</code> 大于或等于 <code>TList</code> 长度，会发生编译期错误
<code>TypeAtNonStrict<TList,idx></code>	传回 <code>TList</code> 某位置（以零为基准）上的型别。如果 <code>index</code> 大于或等于 <code>TList</code> 长度，会传回 <code>NullType</code>
<code>IndexOf<TList,T></code>	传回第一次出现型别 <code>T</code> 的位置（以零为基准）。如果没找到就传回 -1
<code>Append<TList,T></code>	将一个 <code>type</code> 或 <code>typelist</code> 附加到 <code>TList</code> 中
<code>Erase<TList,T></code>	移除 <code>Tlist</code> 内的第一个 <code>T</code> （如果有的话）
<code>EraseAll<TList,T></code>	移除 <code>Tlist</code> 内的所有 <code>T</code> （如果有的话）
<code>NoDuplicates<TList></code>	移除 <code>Tlist</code> 内所有重复的型别
<code>Replace<TList,T,U></code>	以 <code>U</code> 取代 <code>TList</code> 内的第一个 <code>T</code>
<code>ReplaceAll<TList,T,U></code>	用 <code>U</code> 取代 <code>TList</code> 内的所有 <code>T</code>
<code>MostDerived<TList,T></code>	传回 <code>TList</code> 内最深层派生型别（most derived type）。如果没有这样的型别，就传回 <code>T</code> 本身
<code>DerivedToFront<TList></code>	把最深层派生型别（most derived type）移到最前面

4

小型对象分配技术

Small-Object Allocation

译注：本章术语包括 block（区块）、chunk（大块内存；不译）、allocate（分配）、deallocate（归还）、release（释放）、free（释放）。请注意，deallocate 和 release 在本章的意义并不相同。

本章讲述小型对象快速分配器的设计和实现。如果你使用这种分配器来动态分配对象，其所花费的额外开销比起在 stack 内动态分配对象的额外开销显得微不足道。

在不同的场合下，Loki 会用到非常小的对象——甚至小至数个 bytes。第 5 章（Generalized Functors，泛化仿函数）和第 7 章（Smart Pointers，智能指针）广泛运用了小型对象。基于各种原因，多态（polymorphic）行为乃是面向对象编程中最重要的性质，因此这些小型对象不能存储在 stack 内，只能位于 free store（自由空间）中。

C++ 提供 new 和 delete 操作符作为 free store 的主要使用方法。问题是这些操作符都是通用型的，它们的“小型对象分配性能”很糟糕。糟到什么程度呢？分配小型对象时，某些标准的 free store 分配器和本章分配器相比，执行速度会慢一个数量级，内存则耗费两倍之多。

“过早优化是一切罪恶的根源”，Knuth 如是说。但另一方面，按照 Len Lattanzi 的说法：“过时的悲观毫无益处”。对于核心对象如 functors（仿函数）、smart pointers（智能指针）或 strings（字符串）来说，如果它们在执行期产生一个数量级的恶化，对整个项目的成功与否可能带来极大影响。廉价（无额外开销）而快速地动态分配小型对象，好处十分巨大，能让你在运用高级技术的同时，无需担心效能上出现重大损失。因此，探究小型对象在 free store（自由空间）中的分配策略，对程序员具有极大诱惑力。

很多 C++ 书籍，例如 Sutter（2000）和 Meyers（1998a），都谈到了专用型内存分配器的用处。然而 Meyers 将细节“当做一道超难习题留给读者”，Sutter 则将你打发到“你最喜爱的高阶或通用 C++ 编程教本”中。你手上这本书并不指望成为你的最爱，不过本章的确打破砂锅，详尽实现一个奉行 C++ 标准的分配器。

阅读本章之后，你将对内存分配器优化所涉及的某些微妙而有趣的问题有所理解，并知道如何使用 Loki 中责任重大的“小型对象分配器”，以及劳苦功高的 smart pointers 和泛化 functors。

4.1 缺省的 Free Store 分配器

由于某些“神秘原因”，系统缺省的 free store 分配器速度极慢，恶名昭彰。其中一个可能的原因是，它通常只是 C heap 分配器 (malloc/realloc/free) 的浅层包装。C heap 分配器并未特别针对小块内存的分配进行优化。C 程序通常十分有条理地、保守地使用动态内存，决不会采用任何“导致小块内存被大量分配”的手法或技巧。C 程序通常分配中大型对象（数百个或数千个 bytes）。因此，malloc/free 有必要针对小型对象的分配进行优化。

除了速度慢，C++ 缺省分配器的通用性也造成小型对象空间分配的低效。缺省分配器管理一个记忆池 (memory pool)，而这种管理通常需要耗用一些额外内存。一般而言，对于通过 new 配得的每一块内存，其用于簿记管理的部分达到数个 (4~32 个) bytes。如果分配“大小为 1024 bytes”的区块，每一区块的额外开销微不足道 (0.4%~3%)，但如果分配的对象大小为 8bytes，每个对象的额外开销就变成了 50%~400%，令人怵目惊心，尤其是如果你需要大量分配这类小型对象的话。

在 C++ 中，动态分配很重要。执行期多态性 (runtime polymorphism) 和动态分配的联系最为紧密。“Pimpl 手法” (sutter 2000) 就要求“以 free store 分配取代 stack 分配”为前提。

因此，在迈向高效 C++ 程序开发的道路上，缺省分配器的低劣性能成为一种障碍。老练的 C++ 程序员会尽量避免使用“采用 free store 分配行为”的语言构件，因为根据经验他们知道它的成本高昂。缺省分配器不仅是个具体问题，还可能成为一个心理障碍。

4.2 内存分配器的工作方式

研究“内存存在程序中的运用情况”是一项非常有趣的事，这在 Knuth 的划时代著作 (Knuth 1998) 内已经得到证明。Knuth 建立了很多基础的内存分配策略，其后又有更多发明。

内存分配器如何运作？它管理一个由 raw bytes (译注：意指尚未被分配的内存) 所组成的内存池，能够从池中分配任意大小的内存区块。簿记结构可以是像这样的简单控制块：

```
struct MemControlBlock {  
    std::size_t    size_  
    bool          available_  
};
```

MemControlBlock 对象所管理的区块紧接其后，大小为 size_ bytes，然后又是另一个控制块 MemControlBlock，依此类推。

程序开始执行时，内存池起始处只有一个 MemControlBlock，并将所有内存视为一大块来管理。这就是所谓 root 控制块，永不离开最初位置。图 4.1 显示程序刚启动时 1M 内存池的布局。

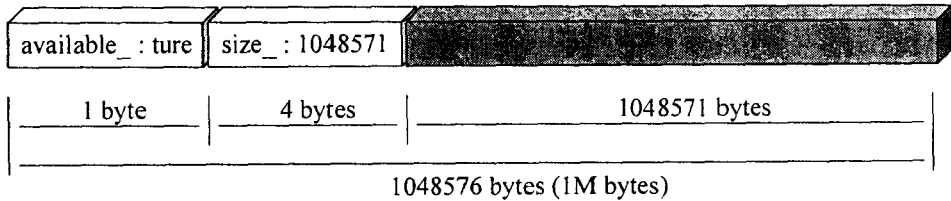


图 4.1 程序启动期 (program startup) 的内存映射图 (memory map)

每次分配申请都会引发对内存的一次线性查找，以求找到一个合适区块（等于或大于申请值）来满足需求。满足需求的策略之多之怪，令人惊讶。你可以使用最先匹配法则（first fit）、最佳匹配法则（best fit）、最差匹配法则（worst fit），甚至随机匹配法则（random fit）。有趣的是最差匹配比最佳匹配好——呃，怎么会有如此的矛盾呢？

每次归还（deallocate）区块，同样需要一次线性搜索，找出待还区块的前一区块并调整其大小。正如你所看到的，这一策略在时间上并非特别高效。但它在空间上的额外开销相当小——对于每一区块，只需额外付出一个 `size_t` 和一个 `bool`。大多数实际场合下你甚至可以将 `size_` 中的一个 bit 挪作 `available_` 之用，从而将 `MemControlBlock` 包捆（pack）至极限：

```
// platform- and compiler- dependent code
struct MemControlBlock {
    std::size_t    size_ : 31;
    bool          available_ : 1;
};
```

如果将指向前一个和指向下一个 `MemControlBlock` 的指针存储在每个 `MemControlBlock` 中，就可以得到常数时间的内存归还动作。这是因为，根据待还区块，我们可以直接取得临近的区块并调整之。这种控制块的结构必然是：

```
struct MemControlBlock {
    bool available_ ;
    MemControlBlock* prev_;
    MemControlBlock* next_;
};
```

图 4.2 显示了这样一种支持 doubly linked list（双向链表）区块的内存池布局。你可以看到其中不再需要 `size_`，因为我们很容易以 `this->next - this` 计算出区块大小。然而每一块分配而得的内存，必须承担两个指针和一个 `bool` 的额外开销。当然你也可以使用因平台而异的某种技巧，将上述的 `bool` 和指针包捆（pack）在一起。

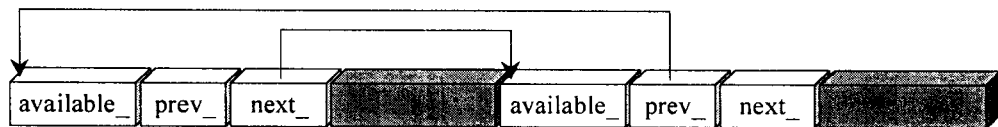


图 4.2 本图所示的分配器可拥有常数时间的区块归还动作

尽管如此，分配动作还是得消耗线性时间。要减轻这样的消耗，有很多巧妙技术可用，每一种技术各有利弊。有趣的是，没有任何一种内存分配策略堪称完美；每一种策略在某种情况下的执行效率都可能比其他方式差。

我们并不需要将“通用型分配器”优化。我们的焦点放在可最佳处理小型对象的“专用型分配器”身上。

4.3 小型对象分配器 (Small-Object Allocator)

本章介绍的小型对象分配器分为四层结构，如图 4.3 所示。下层提供功能供上层使用。最底层是 `Chunk` 型别。每一个 `Chunk` 对象包含并管理一大块内存（所谓 `chunk`），此大块内存本身包含整个固定大小的区块（`block`）。`Chunk` 内含逻辑信息，使用者可根据它来分配和归还区块。当 `chunk` 中不再剩余 `blocks` 时，分配失败并传回零。

第二层是 `FixAllocator class`，其对象以 `Chunk` 为构件。`FixAllocator` 主要用来满足那些“累计总量超过 `Chunk` 容量”的内存分配请求。`FixAllocator` 会通过一个 `array`（译注：其实是个 `vector`）将 `Chunks` 组合起来以达到目的。如果出现一笔新申请，但现有的 `Chunks` 都被占用了，此时 `FixAllocator` 会产生（分配）一块新 `Chunk`，并将它添入 `array`（译注：其实是 `vector`）内，再由新的 `Chunk` 满足需求。

第三层 `SmallObjAllocator` 提供的是通用性的分配/归还函数。此物拥有数个 `FixedAllocator` 对象，每一个负责分配某特定大小的对象。根据“申请的 `bytes` 个数”不同，`SmallObjAllocator` 对象会将内存分配申请分发给辖下某个 `FixedAllocator`。但如果请求量过大，会转发给缺省的（系统提供的）`::operator new`。

最后一层是 `SmallObject`，它包装 `FixedAllocator`，以便向 C++ classes 提供封装良好的分配服务。`SmallObject` 重载 `operator new` 和 `operator delete`，将任务转给 `SmallObjAllocator` 对象去完成。采用这种方法，你可以让你的对象享受专用分配器的好处，而这一切只需你的对象从 `SmallObject` 派生出来就可以办到。

你也可以直接使用 `SmallObjAllocator` 和 `FixedAllocator`（`Chunk` 则过于原始，而且不够安全，因而被定义在 `FixedAllocator` 的 `private` 区段内），但大多数情况下客户端代码只需派生自 `SmallObject` 就可以执行高效率分配动作。这是一个十分简单易用的接口。

SmallObject	* 提供对象层次 (object level) 的服务 * 通透性——classes 只需继承自 SmallObject 即可享受服务
SmallObjAllocator	* 能够分配多种不同大小的小型对象 * 可根据参数进行配置 (configurable)
FixedAllocator	* 分配特定大小的对象
Chunk	* 根据某个特定大小来分配对象 * 对象分配数量的上限是固定的

图 4.3 小型对象分配器 (small object allocator) 的四层构造

4.4 Chunks (大块内存)

每一个 chunk 对象包含并管理一大块内存 (chunk)，其中包含固定数量的区块 (blocks)。你可以在构造期间设定区块的大小和数量。

Chunk 包含逻辑信息，让你得以从该大块内存中分配和归还区块。一旦 chunk 之中没有剩余区块，分配函数便传回零。

Chunk 定义如下：

```
// Nothing is private - Chunk is a Plain Old Data (POD) structure
// structure defined inside FixedAllocator
// and manipulated only by it
struct Chunk
{
    void Init(std::size_t blockSize, unsigned char blocks);
    void Release();
    void* Allocate(std::size_t blockSize);
    void Deallocate(void* p, std::size_t blockSize);
    unsigned char* pData_;
    unsigned char
        firstAvailableBlock_,
        blocksAvailable_;
};
```

除了以一个指针指向被管理之内存本身，chunk 还保存以下整数值：

- firstAvailableBlock_，chunk 内的第一个可用区块的索引号
- blocksAvailable_，chunk 内的可用区块总数

Chunk 的接口非常简单。Init() 用于初始化, Release() 用来释放已配得的内存。Allocate() 用来分配一个区块, Deallocate() 用来归还某个区块。你必须传给 Allocate() 和 Deallocate() 一个区块大小值, 因为 Chunk 不保存它 (译注: 如果是 system new, 就会以所谓 "cookie" 保存它)。区块的大小在较高层级中已知, 如果此处多设一个 blockSize_ 成员, 会给 Chunk 造成空间和时间上的浪费——不要忘记我们现在处于最底层, 每件事情每样东西都至关重要。基于效率上的考虑, Chunk 并未定义构造函数、析构函数和 assignment (赋值) 操作符。在这一层定义自己的 copy 语义会损及上一层的效率——上一层将 chunks 存储于一个 vector 内。

Chunk 结构反映出设计中的一个重要折衷。由于 blocksAvailable_ 和 firstAvailableBlock_ 都是 unsigned char 型别, 因此一个 Chunk 在一部 8-bit char 机器上无法拥有 255 个以上的区块。很快你就会看到, 这个决定还不赖, 可帮你省去许多头疼的事。

现在看看最有趣的部分。区块(s) 若非正被使用, 就是尚未被使用。未被使用的区块当然可以拿来存储任何东西。是的, 我们要善用这一点, 拿“未被使用的区块”的第一个 bytes 来放置“下一个未被使用的区块”的索引号。由于 firstAvailableBlock_ 已经持有第一个可用区块的索引号, 因此我们便有了一个由“可用区块”组成的完整单向链表 (singly linked list), 无须占用额外内存。

初始化时, Chunk 对象看起来像图 4.4 那样。初始化函数如下:

```
void Chunk::Init(std::size_t blockSize, unsigned char blocks) {
    pData_ = new unsigned char[blockSize * blocks];
    firstAvailableBlock_ = 0;
    blocksAvailable_ = blocks;
    unsigned char i = 0;
    unsigned char* p = pData_;
    for (; i != blocks; p += blockSize) {
        *p = ++i;
    }
}
```

融合于这一数据结构内的单向链表 (singly linked list) 真是个好东西。它提供了一种快速、高效的方法来寻找这一大块内存中的可用区块, 却不占用额外空间。在 Chunk 内分配和归还区块时, 之所以只消耗常数时间, 得特别感谢这一内嵌单向链表。

现在你应该可以明白为什么我要将区块数量限制在 unsigned char 的大小了。假设我们使用一个较大型别, 例如 unsigned short (它在很多机器上是 2 bytes), 我们将遭遇两个问题, 一个是**大问题, 一个是小问题。

- 我们无法分配小于 sizeof(unsigned short) 的区块, 这真令人尴尬, 因为我们现在正打算建立一个小型对象分配器。这是我所谓的小问题。
- 我们会遇到齐位 (alignment) 问题。假设你为一个 5 bytes 区块建立一个专属分配器。这种情况下如果想将“指向如此一个 5 bytes 区块”的指针转换为 unsigned int, 会造成不确定 (未定义) 的行为。这是我所谓的大问题。

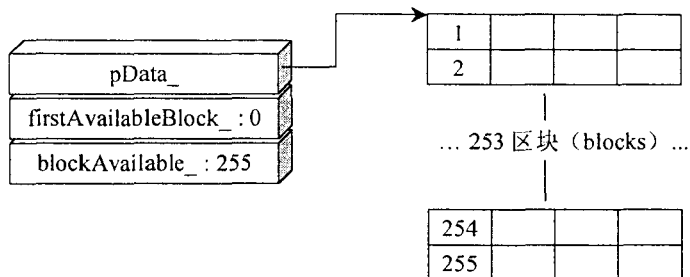


图 4.4 一个内含 255 个区块的 chunk，每个区块大小都是 4 bytes

解决办法很简单：以 `unsigned char` 作为这个“特异索引号”的数据型别。根据定义，`char` 的大小为 1，无齐位问题，因为即使指向 `raw memory` 的指针也是指向 `unsigned char`。

这个设计会限制 `chunk` 所含区块（blocks）的最大数量。`chunk` 所拥有的区块数将无法多于 `UCHAR_MAX`（其值在大多数系统中为 255）。这一点可以接受，即使区块真的非常小，例如 1~4 bytes。对于较大区块，这个限制没什么差别，毕竟我们不想分配太大的 `chunks`。

分配函数 `Allocate()` 的动作是取出 `firstAvailableBlock_` 所代表的区块，然后调整 `firstAvailableBlock_`，使指向下一个可用区块。这是典型的 `list` 操作。

```
void* Chunk::Allocate(std::size_t blockSize)
{
    if (!blocksAvailable_) return 0; // 译注：一个比较动作
    unsigned char* pResult =          // 译注：一个赋值动作
        pData_ + (firstAvailableBlock_ * blockSize); // 译注：一个索引访问
    // Update firstAvailableBlock_ to point to the next block
    firstAvailableBlock_ = *pResult; // 译注：一个赋值动作和一个提领动作
    --blocksAvailable_;             // 译注：一个递减动作
    return pResult;
}
```

`Chunk::Allocate()` 的成本是：一个比较动作、一个索引访问、一个提领（`dereference`）动作、两个赋值动作、一个递减动作；成本很小。最重要的是不需查找动作。目前为止一切良好。图 4.5 显示第一次区块分配完成后，`Chunk` 对象的布局。

归还函数 `Deallocate()` 的行为完全相反：将区块传回给自由列表（`free list`），然后累加 `blocksAvailable_`。不要忘记，由于 `Chunk` 对区块大小一无所知，所以你必须将区块大小当做参数传给 `Deallocate()`。

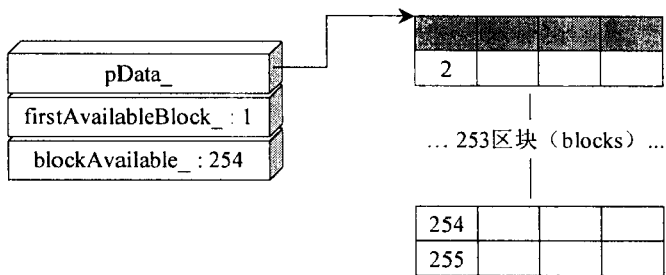


图 4.5 chunk 经过一次分配后的情况，被分配的区块以灰色呈现

```

void Chunk::Deallocate(void* p, std::size_t blockSize)
{
    assert(p >= pData_);
    unsigned char* toRelease = static_cast<unsigned char*>(p);
    // Alignment check
    assert((toRelease - pData_) % blockSize == 0);
    *toRelease = firstAvailableBlock_;
    firstAvailableBlock_ = static_cast<unsigned char*>(
        (toRelease - pData_) / blockSize);
    // Truncation check
    assert(firstAvailableBlock_ ==
        (toRelease - pData_) / blockSize);
    ++blocksAvailable_;
}

```

上述的归还函数很精练，但有很多 assertions（而且尚不能够考虑所有出错状态）。Chunk 秉承了 C 和 C++ 在内存分配上的重要传统：准备面对最坏情况——也就是你将错误指针传给 Chunk::Deallocate()。

4.5 大小一致 (Fixed-Size) 的分配器

小型对象分配器的第二层由 FixedAllocator 构成。此物知晓如何分配和归还“特定大小的区块”，其大小不受限于 chunk，只受限於系统可用内存量。

为达到这一点，FixedAllocator 将 Chunk 对象集合存放到一个 vector 中。任何时候若出现分配请求，FixedAllocator 便找出一个适当的 Chunk 企图满足之。如果所有 Chunk 都用光了，FixedAllocator 会添一个新 Chunk。下面是 FixedAllocator 定义式中的相关部分：

```

class FixedAllocator
{
    ...
private:
    std::size_t          blockSize_;
    unsigned char        numBlocks_;
    typedef std::vector<Chunk> Chunks;
    Chunks chunks_;
}

```

```

    Chunk* allocChunk_;
    Chunk* deallocChunk_;
};

```

为提高查找速度，面对每一次分配，`FixedAllocator` 并非遍历整个 `chunks_` 寻找空间。它保存一个指针 `allocChunk_` 指向“最近一次分配所使用的 `chunk`”。任何时候只要出现分配请求，`FixedAllocator::Allocate()` 便首先检查 `allocChunk_`，看看是否有可用空间。如果 `allocChunk_` 尚有可用空间，分配请求将藉由 `allocChunk_` 瞬间获得满足。否则就会引发一次线性查找（甚至可能会有一个新 `Chunk` 添加到 `chunks_ vector` 中）。无论是哪一种情况，`allocChunk_` 都会更新，指向刚找到的或新添加的 `chunk`。采用这种方法可以增加“提高下次分配速度”的可能性。以下代码实作出上述算法：

```

void* FixedAllocator::Allocate()
{
    if (allocChunk_ == 0 || allocChunk_->blocksAvailable_ == 0) {
        // No available memory in this chunk
        // Try to find one
        Chunks::iterator i = chunks_.begin();
        for (;;) {
            if (i == chunks_.end()) {
                // All filled up - add a new chunk
                chunks_.push_back(Chunk());
                Chunk& newChunk = chunks_.back();
                newChunk.Init(blockSize_, numBlocks_);
                allocChunk_ = &newChunk;
                deallocChunk_ = &chunks_.front();
                break;
            }
            if (i->blocksAvailable_ > 0) {
                // Found a chunk
                allocChunk_ = *i;
                break;
            }
        }
    }
    assert(allocChunk_ != 0);
    assert(allocChunk_->blocksAvailable_ > 0);
    return allocChunk_->Allocate(blockSize_);
}

```

运用这个策略，`FixedAllocator` 可以在常数时间内满足大多数分配请求，只偶尔会因为查找或添加新区块而变慢。某些特殊的内存分配状况会使上述策略效率不高，但现实世界中那种情况并不频繁。别忘了，每一种分配器都有弱点，就像阿契里斯 (Achilles) 的脚踵一样（译注：希腊神话中的阿契里斯，除了脚踵，浑身刀枪不入）。

内存归还 (deallocation) 比较棘手, 因为归还的时候缺少一则信息——我们拥有的只是一个指针指向待还区块, 我们不知道那个指针属于哪个 chunk。是的, 我们可以遍历 `chunks_`, 检查指针是否落在 `pData_` 和 `pData_ + blockSize_ * numBlocks_` 之间。如果是, 就将指针传给那个 chunk 的 `Deallocate()`。问题是这么做很耗时间。虽然分配速度很快 (常数时间), 归还却需耗用线性时间, 这不好。我们需要用另一种方法来提高归还速度。

我们可以为已归还的区块增设一个高速缓存 (cache) 存储区。当客户端程序通过 `FixedAllocator::Deallocate(p)` 归还一个区块时, `FixedAllocator` 并不将 `p` 传回给对应的 chunk, 而是将 `p` 添加到一块内部存储区 (那是一块用以保存可用区块的高速缓存设备) 内。一旦出现新的分配请求, `FixedAllocator` 首先在高速缓存区中查找。如果高速缓存区不为空, 就立即从中取出一个可用指针, 这是速度很快的操作。只有当高速缓存区用罄时, `FixedAllocator` 才走标准途径, 将分配请求转给一个 chunk。这是一个很有价值的策略, 但对于小型对象经常使用的某些分配和归还状况而言, 它表现得并不理想。

分配小型对象时有四种主要倾向:

- 批量分配。一次分配很多小型对象。当你初始化一群指针, 而它们都指向小型对象时, 就属于这种情况。
- 以相同次序归还。很多小型对象的归还次序和分配次序相同。大多数 STL 容器被摧毁时就会发生这种情况¹²。
- 以相反次序归还。很多小型对象的归还次序和分配次序相反。当你在 C++ 程序中调用“操作小型对象”的函数时, 这种情况很自然会发生。函数引数和暂时性 stack 变量都属此类。
- 蝶式 (Butterfly) 分配和归还。对象的生成和销毁不遵循一定的顺序。当你的程序正在运行并偶尔需要小型对象时, 这种情况会发生。

高速缓存 (caching) 策略非常适合“蝶式分配和归还”情况, 因为如果分配和归还随机发生, 这种策略能使它们持续保持快速。但对于批量分配和归还, 高速缓存设备无甚帮助, 更糟的是它甚至会降低归还速度, 因为高速缓存设备本身的清理也需要时间¹³。

一个较好的策略是采用与分配相同的概念。成员变量 `FixedAllocator::deallocChunk_` 指向归还动作所用的最后那个 chunk 对象。任何时候只要发生归还动作, 首先便检查 `deallocChunk_`。然后, 如果那是个错误的 chunk, `Deallocate()` 会执行一次线性搜索, 满足需求并更新 `deallocChunk_`。

¹² 标准 C++ 并未定义标准容器内的对象析构次序, 因此每一位实作人员都需要自己抉择。容器往往经由 forward 迭代器被摧毁。然而某些实作者会选择比较“自然”的次序: 他们会以相反次序来摧毁对象。基本理由是 C++ 对象以其生成次序的相反次序被摧毁。

¹³ 我无法举一个适当的高速缓存 (caching) 方案来讨论它在“同序归还”和“逆序归还”是否一样好。永远是鱼与熊掌不可兼得。由于两种倾向都很可能发生, 所以高速缓存其实不是个好选择。

针对先前所列的分配情况,有两个重要调整可以提高 `Deallocate()` 的速度。首先 `Deallocate()` 从 `deallocChunk_` 附近开始查找合适的 `Chunk`。也就是说 `chunks_` 的查找是从 `deallocChunk_` 开始,以两个迭代器 (iterators) 分别向上和向下进行。对于以正序或逆序进行的批量归还动作,这可以大大提高速度。批量分配期间 `Allocate()` 按序添加 `chunks`, 归还期间要么立即发现 `deallocChunk_` 就是目标,要么就在下一步找到正确的 `chunk`。

第二个调整是避免边界条件的发生。假设 `allocChunk_` 和 `deallocChunk_` 都指向 `vector` 的最后一个 `Chunk`, 而该 `chunk` 已无剩余空间。那么,假设执行以下代码:

```
for (...)
{
    // Some smart pointers use the small-object
    // allocator internally (see Chapter 7)
    SmartPtr p;
    ... use p ...
}
```

循环的每一次迭代都会生成 (而后销毁) 一个 `SmartPtr` 对象。生成时由于没有更多内存, `FixedAllocator::Allocate()` 会产生一个新 `Chunk`, 并将它添加到 `chunks_ vector` 中。销毁时 `FixedAllocator::Deallocate()` 会检测到一个空区块并将它归还。每进行一次迭代,上述昂贵过程便重复一次。

这样的低效让人无法接受。因此在归还过程中,只有当“存在两个空 `chunk` 时”,其中一个 `chunk` 才会被释放。如果只有一个空 `chunk`,它会被高效地置换至 `chunk_vector` 尾部。这样我们便能避免昂贵的 `vector<Chunk>::erase()` 动作,因为我们永远只需删除最后一个元素。

当然,某些情形会使上述简单的设想失效。如果你在循环内分配一个 `vector`, 每个元素都是 `SmartPtr`, 具有合适大小,那么你会回到老问题上,但是这种情形较少出现。而且就像本章简介所提,任何分配器都可能在某个特定情况下表现得比其他分配器差。

这样的归还策略也适合蝶式分配 (butterfly allocation)。纵使不按一定次序分配数据,程序也具有某种地域性 (locality) 倾向。也就是说它们一次只访问少量数据。指针 `allocChunk_` 和 `deallocChunk_` 可以良好处理这种情况,因为它们就像“最近一次分配和归还”的高速缓存设备。

结论是,我们现在有了一个 `FixedAllocator` class, 能够满足特定大小的区块分配请求,速度和内存运用效率都令人满意,并且针对小型对象分配的典型情况进行了优化。

4.6 SmallObjAllocator Class

本章分配器分层架构中的第三层是 `SmallObjAllocator`, 这是个 class, 能够分配任意大小的对象。`SmallObjAllocator` 藉由“聚集数个 `FixedAllocator` 对象”来达到这一服务。当 `SmallObjAllocator` 收到一个分配请求时,也许将该分配请求转给最佳匹配的 `FixedAllocator`, 要么就转给缺省的 `::operator new`。

下面是 `SmallObjAllocator` 的概貌，文字说明列于代码之后。

```
class SmallObjAllocator
{
public:
    SmallObjAllocator(
        std::size_t chunkSize,
        std::size_t maxObjectSize);
    void* Allocate(std::size_t numBytes);
    void Deallocate(void* p, std::size_t size);
    ...
private:
    std::vector<FixedAllocator> pool_;
    ...
};
```

上述构造函数接受两个参数，用以对 `SmallObjAllocator` 进行配置设定 (configure)。第一参数 `ChunkSize` 代表 `chunk` 的缺省大小 (Chunk 对象的长度皆以 bytes 计算)，第二参数 `maxObjectSize` 是所谓“小型对象”的最大认可值——所有小型对象皆需小于此值。如果申请的区块大小超过 `maxObjectSize`，`SmallObjAllocator` 会将请求转给 `::operator new`。

奇怪的是 `Deallocate()` 有一个参数用来表示“待归还大小”。这么做是为了让分配更快，否则这个函数就不得不查找 `pool_` 中的所有 `FixedAllocator`，以期找到第一参数 (一个指针) 所属的那个 `FixedAllocator`。这成本太高了，所以 `SmallObjAllocator` 要求你传入待归还区块的大小。下一节你会看到，这一任务由编译器优雅地处理掉了。

`FixedAllocator` 区块的大小和 `pool_` 之间有什么映射关系？换句话说，如果给出一个大小，哪个 `FixedAllocator` 会负责这类区块的分配和归还任务？

一个简单而且高效的做法是，让 `pool_[i]` 处理大小为 `i` 的对象。首先初始化 `pool_`，使其大小为 `maxObjectSize`，然后初始化每一个相应的 `FixedAllocator`。一旦接到 `numBytes` 分配请求，`SmallObjAllocator` 便将该请求转给 `pool_[numBytes]` (引发一个常数时间的操作)，抑或转给 `::operator new`。

然而，这个方案并不如看上去那么巧妙。“有效果”并不总是意味着“有效率”。问题在于你可能只需少量分配器，用以分配特定大小的对象 (具体情况取决于你的应用程序)。例如也许你只需产生 4 bytes 和 64 bytes 两种对象，再没其他的了。但是这种情况下你还是得为 `pool_` 分配 64 个或更多个元素，虽然你只使用其中两个。

齐位 (alignment) 和填补 (padding) 会进一步造成 `pool_` 的空间浪费。很多编译器会为所有“客户自定义型别”进行填补，使其大小为某数 (2, 4, 或更大) 的倍数。如果编译器将所有结构都填补为 4 的倍数，你就会只用到 `pool_` 的 25%，其余都被浪费了。

因此，比较好的做法是：为节约内存而牺牲一点点查找速度¹⁴。只有“某种大小的分配需求”至少发生一次，我们才存储对应的 `FixedAllocator`。采用这种方法，`pool_` 就可以容纳各种对象大小而不需要成长太多。如果要提高查找速度，可以将 `pool_` 按区块大小排序

为改善查找速度，我们可以采用 `FixedAllocator` 所采用的相同策略。让 `SmallObjAllocator` 保存两个指针，分别指向最近一次分配或最近一次归还所用的 `FixedAllocator`。以下完整列出 `SmallObjAllocator` 的成员变量：

```
class SmallObjAllocator {
    ...
private:
    std::vector<FixedAllocator> pool_;
    FixedAllocator* pLastAlloc_;
    FixedAllocator* pLastDealloc_;
};
```

一旦发生分配请求，首先检查 `pLastAlloc_`。如果大小不对，`SmallObjAllocator::Allocate()` 会在 `pool_` 身上执行二分查找（binary search）。归还请求亦以类似方式处理，唯一区别是 `SmallObjAllocator::Allocate()` 可以在 `pool_` 中插入一个新的 `FixedAllocator` 对象。

就像先前对 `FixedAllocator` 的讨论一样，这个简单的高速缓存（caching）方案对于批量分配和归还很有效，其操作效率为常数时间。

4.7 帽子下的戏法

本章小型对象分配器的第四层结构为 `SmallObject`。这是个 base class，将 `SmallObjAllocator` 提供的功能做更方便运用的包装。

`SmallObject` 重载了系统提供的 `operator new` 和 `operator delete`。这么一来，只要你生成一个 `SmallObject` 派生对象，重载后的行为便会加入整体行动之中，于是将分配请求发送给前述的“定量分配器”。`SmallObject` 的定义非常简短，只不过情节可能有点复杂：

```
class SmallObject {
public:
    static void* operator new(std::size_t size);
    static void operator delete(void* p, std::size_t size);
    virtual ~SmallObject() {}
};
```

看起来非常简短，但有些小地方要注意。很多 C++ 书籍（例如 Sutter 2000）告诉我们：如果想在 class 中重载系统缺省的 `operator delete`，就必须以一个“指向 void 的指针”作为 `operator delete` 的唯一参数。

¹⁴ 现代系统中，你可以在使用较少内存的同时，寄希望于速度的增加。这是因为主内存（main memory）和高速缓存内存（cache memory）之间有很大的差异：前者量大而慢，后者量小而快。

这是 C++ 的一个漏洞，我们对此漏洞很感兴趣（请回忆一下，我们设计 `SmallObjAllocator` 时是将“待归还区块之大小”设计为一个参数）。在标准 C++ 中，你其实可以藉由两种方式重载系统缺省的 `operator delete`，一则是这样：

```
void operator delete(void* p);
```

再则是这样：

```
void operator delete(void* p, std::size_t size);
```

Sutter (2000) p.144 对此曾有详尽的讨论。

如果采用第一形式，意味着你不在意“待释区块之大小”。但事实上我们非常需要知道这个区块大小，才能将它传给 `SmallObjAllocator`。所以 `SmallObject` 采用上述第二形式来重载 `operator delete`。

编译器如何自动提供对象大小呢？看起来似乎无可避免得增加每个对象的内存额外开销，而额外开销却正是本章希望避免的。

不，根本没有任何额外开销！请看以下代码：

```
class Base {
    int a_[100];
public:
    virtual ~Base() {}
};
class Derived : public Base {
    int b_[200];
public:
    virtual ~Derived() {}
};
...
Base* p = new Derived;
delete p;
```

`Base` 和 `Derived` 大小不同。为避免因存储“`p` 所指对象的大小”而带来额外开销，编译器玩了个花招：它即时产生一些代码用以计算对象大小。有四种技术可以做到这一点，一一列出于下。（偶而转换角色戴上编译器设计者的帽子，不失为一件趣事，转眼之间你可以玩些不可思议的小把戏，而一般程序员却无法做到◎）

1. 将一个 `Boolean` 标志传给析构函数，表示“销毁对象之后要不要调用 `operator delete`”。`Base` 拥有一个 `virtual` 析构函数，因此本例中的 `delete p` 会作用于正确对象（亦即 `Derived`）上。彼时对象大小可以静态获得（亦即 `sizeof(Derived)`），编译器只需将该值传给 `operator delete` 即可。
2. 让析构函数传回对象大小。你可以要求每一个析构函数在销毁对象之后传回 `sizeof(Class)`（别忘了你现在是编译器设计者）。这一方案之所以有效是因为 `Base` 拥有一个 `virtual` 析构函数。调用析构函数后，C++ runtime（执行期系统）会调用 `operator delete` 并将析构函

数回传值传给它。

3. 实作出一个隐藏的 `virtual` 成员函数，例如 `_Size()`，用以获得对象大小。而后 C++ runtime（执行期系统）调用该函数并保存结果，销毁对象，调用 `operator delete`。这个做法看起来似乎缺乏效率，优点是 `_Size()` 可另做它用。
4. 在每个 class 的虚函数表（virtual function table, vtable）某处直接保存大小。这个做法既灵活又高效，但技术上比较困难。

（现在请摘下编译器设计者的帽子）如你所见，为了向你的 `operator delete` 传递正确大小值，编译器费了很大工夫。那么，每次归还对象时，何必忽略此值又执行一次昂贵的查找呢？

这一切都配合得井井有条：`SmallObjAllocator` 需要知道“待归还区块的大小”，编译器提供其值，`SmallObject` 将该值转给 `FixedAllocator`。

以上大多数方案都假设你为 `Base` 定义了一个 `virtual` 析构函数。这再次说明将多态性（polimorphic）classes 的析构函数定义为 `virtual` 是多么重要。如果你没这么做，万一你 `delete` 一个 `base class` 指针，而该指针实际却指向一个 `derived class` 对象，就会造成不确定（未定义）行为。就本章分配器而言，这会使程序在调试模式下因为 `assertion` 而中断执行，在非调试模式下则造成崩溃。喔，任何人都都会同意这种行为的确属于“不确定”范畴☹。

为了让你不至于总是必须记住这一切，也为了避免将时间浪费于“不这么做因而导致的没日没夜的调试”，`SmallObject` 定义了一个 `virtual` 析构函数。从 `SmallObject` 派生出来的任何 classes 都会继承这个 `virtual` 析构函数。这同时也将我们带到“`SmallObject` 的实作”话题上。

对整个程序而言，我们只需要唯一一个 `SmallObjAllocator`，它必须被正确地构造并被正确地摧毁。这对它本身而言是个棘手难题。幸运的是通过 `SingletonHolder template`，Loki 完全解决了这个问题。`SingletonHolder template` 将于第 6 章讨论（让读者跳阅后继章节实在是很遗憾，但如果浪费这一“复用 `SingletonHolder`”的好机会恐怕我会更遗憾）。眼下请暂时先将 `SingletonHolder` 视为一个设备（device），借助这个设备我们可以高阶管理“某个 class 的唯一实体”。假设 class 名为 `x`，我们可以通过 `SingletonHolder<x>` 将它具现化（`instantiate`），而后可调用 `SingletonHolder<x>::Instance()` 取得该唯一实体。关于 `Singleton` 设计模式，Gamma 四人著作（1995）中有详尽阐述。

有了 `SingletonHolder`，`SmallObject` 的实作就极为简单了：

```
typedef SingletonHolder<SmallObjAllocator> MyAlloc;
void* SmallObject::operator new(std::size_t size) {
    return MyAlloc::Instance().Allocate(size);
}
void SmallObject::operator delete(void* p, std::size_t size) {
    MyAlloc::Instance().Deallocate(p, size);
}
```

4.8 简单，复杂，终究还是简单

`SmallObject` 的实作十分简单，但如果将多线程 (multithreading) 纳入考量，就不可能那么简单。是的，上述唯一一个 `SmallObjAllocator` 实体 (对象) 被所有 `SmallObject` 实体 (对象) 共享。如果那些实体属于不同的线程，我们实际上就是在多线程之间共享这个 `SmallObjAllocator`。正如本书附录所言，这种情况下我们必须采取特殊对策。我们似乎得回到小型对象分配器的各层实作细节中，找出关键操作，适当增加锁定 (locking) 机制。

毋庸置疑，多线程确实会给事情带来一定的复杂度，但还不至于太过复杂，因为 `Loki` 已经定义了高阶的对象同步机制 (synchronization)。迈向复用 (reuse) 的最好途径就是真正用它，因此我含入 `LokiThreads.h`，并对 `SmallObject` 作如下修改 (粗体即修改之处)：

```
template <template <class T> class ThreadingModel>
class SmallObject : public ThreadingModel<SmallObject>
{
    ... 如同以往 ...
};
```

`operator new` 和 `operator delete` 的定义也需稍作改动：

```
template <template <class T> class ThreadingModel>
void* SmallObject<tm>::operator new(std::size_t size)
{
    Lock lock;
    return MyAlloc::Instance().Allocate(size);
}

template <template <class T> class ThreadingModel>
void SmallObject<tm>::operator delete(void* p, std::size_t size)
{
    Lock lock;
    MyAlloc::Instance().Deallocate(p, size);
}
```

就这样！不需对底层做任何修改——有了高阶锁定 (locking) 机制，它们的功能完全获得保护。

此处对 `Loki` 所提供的“单件 (Singleton) 管理”和“多线程特性”的运用，证实了“复用”的强大威力。“全局变量的生命期”和“多线程”两个题目各有复杂度，如果单纯从基本原理出发，试图在 `SmallObject` 中处理这些问题，将会苦不堪言。是的，心平气和地想象一下，如果你正如火如荼地实作 `FixedAllocator` 的复杂高速缓存 (caching) 功能，却遇上“多个线程将同一对象初始化 (这当然是错误的)”的情况… 喔，天啊！

4.9 使用细节

本节讨论如何在应用程序中使用 `SmallObj.h`。

为了使用 `SmallObject`，你必须为 `SmallObjAllocator` 构造函数提供适当的参数：`chunk` 大小和小型对象的最大尺寸。何谓小型对象？对象多小才算是小型对象呢？

为了回答这个问题，让我们回头看看生成小型对象的目的：我们希望降低“系统缺省分配器”附带的空间和时间上的额外开销。

“系统缺省分配器”带来的空间额外开销在不同情况下差异很大，毕竟它也有可能采取类似本章所讨论的改善策略。然而对大多数通用型分配器而言，每个对象的开销在典型桌面系统上大约是 4~32 bytes。如果额外开销为 16 bytes，对一个 64 bytes 对象的浪费率便是 25%，唔，很高。因此 64 bytes 对象应该被视为小型对象。

如果你让 `SmallObjAllocator` 处理太大对象，你就会分配出比需求量多得多的内存——别忘了，即使你释放所有小型对象，`FixedAllocator` 还是会保留一个 `chunk`。

Loki 允许你选择，也为你准备了适当的缺省值。`SmallObj.h` 中有三个预处理符号（preprocessor symbols），如表 4.1 所示。面对项目中的所有源代码，你应该使用同一个预处理符号进行编译（或干脆不定义它们而使用缺省值）。如果不这么做，也没什么大不了，只不过会产生出更多不同大小的 `FixedAllocators`。

缺省值是针对“具有合理内存容量”的桌面系统而设的。如果你将 `MAX_SMALL_OBJECT_SIZE` 或 `DEFAULT_CHUNK_SIZE` 两者中的任一个定义（`#define`）为零，`SmallObj.h` 会经由条件编译产出“只采用系统缺省之 `::operator new` 和 `::operator delete`”的代码，不带任何额外开销。对象接口没变，但其函数为 `inline` 函数，而内存请求将被转至系统缺省的 `free store`（自由空间）分配器身上。

`class template SmallObject` 原本只有一个参数。为支持不同的 `chunk` 大小和对象大小，如今另带两个 `template` 参数，分别缺省为 `DEFAULT_CHUNK_SIZE` 和 `MAX_SMALL_OBJECT_SIZE`：

```
template
<
    template <class T>
        class ThreadingModel = DEFAULT_THREADING,
        std::size_t chunkSize = DEFAULT_CHUNK_SIZE,
        std::size_t maxSmallObjectSize = MAX_SMALL_OBJECT_SIZE
    >
class SmallObject;
```

因此，如果使用 `SmallObject<>` 形式，你将获得一个 `class`，可在缺省的多线程模式下运作，并具有内存管理方面的缺省选择。

表 4.1 SmallObj.h 的预处理符号 (preprocessor symbols)

符号	意义	缺省值
DEFAULT_CHUNK_SIZE	chunk 缺省大小 (单位: byte)	4096
MAX_SMALL_OBJECT_SIZE	SmallObjAllocator 所处理的 最大型“小型对象”	64
DEFAULT_THREADING	应用程序所使用之缺省线程模式。 多线程程序应该将此符号定义为 ClassLevelLockable	继承自 Thread.h

4.10 摘要

某些 C++ 技术亟需运用“来自于 free store (自由空间) 的小型对象”，这是因为 C++ 的执行期多态性 (runtime polymorphism) 离不开“动态分配”和“pointer/reference 语义”。但系统缺省的分配器 (以全局的 `::operator new` 和 `::operator delete` 呈现) 通常只针对大型对象 (而非小型对象) 的分配进行优化，造成“缺省分配器不适合用来分配小型对象”的现象，因为那会导致速度变慢，而且每个小型对象的内存额外开销也难以忽视。

解决方案是使用专用型小型对象分配器，这是一种经过优化的分配器，专门用来处理小区块 (数十个或数百个 bytes) 分配。小型对象分配器使用 chunks (大块空间)，并运用独特方式将 chunks 组织起来以减少空间和时间上的损失。C++ 执行期系统有助于达成这一目标，因为它可以提供“待还区块”的大小——只要运用较少人知的一种 `operator delete` 重载形式，就可以获得该大小值。

Loki 的小型对象分配器达到“最大可能速度”了吗？没有！Loki 分配器只是在标准 C++ 的限定范围内工作。正如你在本章所见，面对诸如齐位 (alignment) 这样的问题，我们必须保守处理，而保守意味着“未达到最佳状态”。但 Loki 分配器的确已经相当快速、简单和稳定，而且具有可移植的优点。

4.11 小型对象分配器 (Small-Object Allocator) 要点概览

- Loki 所实现的分配器有四层结构。第一层由 private type `Chunk` 组成，它将相等大小的内存组织为一个个大块 (chunks)。第二层为 `FixedAllocator`，使用一个“具可变长度的 vector”来管理 chunks，以满足分配需求，使分配总量可至“整个系统的可用内存”。第三层 `SmallObjAllocator` 运用多个 `FixedAllocator` 对象，得以分配任意大小的对象，其中对小型对象的分配系通过 `FixedAllocator` 完成，对大型对象的分配则转由 `::operator new` 完成。最后的第四层由 `SmallObject` 担纲，这个 class template 对 `SmallObjAllocator` 采用进一步包装。

- `SmallObject` class template 大致定义如下:

```
template
<
    template <class T>
        class ThreadingModel = DEFAULT_THREADING,
        std::size_t chunkSize = DEFAULT_CHUNK_SIZE,
        std::size_t maxSmallObjectSize = MAX_SMALL_OBJECT_SIZE
>
class SmallObject
{
public:
    static void* operator new(std::size_t size);
    static void operator delete(void* p, std::size_t size);
    virtual ~SmallObject() {}
};
```

- 只要继承自 `SmallObject`, 你的 class 就可以尽享本章小型对象分配器带来的好处。你可以藉由缺省参数 (`SmallObject<>`) 来具现化 (instantiated) `SmallObject` class template, 也可以调整其线程模式或内存分配参数。
- 如欲在多个线程中通过 `new` 生成对象, 你必须为上述的 `ThreadingModel` 参数选定某个多线程模式。本书附录对 `ThreadingModel` 有更多介绍。
- `DEFAULT_CHUNK_SIZE` 缺省为 4096。
- `MAX_SMALL_OBJECT_SIZE` 缺省为 64。
- 你可以运用 `#define` 定义 `DEFAULT_CHUNK_SIZE` 或 `MAX_SMALL_OBJECT_SIZE` (或两者), 从而造成其缺省值被忽略。这些宏 (macros) 被展开后必须是型别为 `std::size_t` 的常数, 或是可转换为 `std::size_t` 的常数。
- 如果将 `DEFAULT_CHUNK_SIZE` 或 `MAX_SMALL_OBJECT_SIZE` 定义为零, `SmallObj.h` 会通过条件编译方式产生代码, 使分配任务直接转给系统缺省的 `free store` (自由空间) 分配器, 接口维持不变。万一你想比较“专用内存分配器”使用前后对程序带来的影响, 这就很有用。

第二篇

组件

Components

5

泛化仿函数

Generalized Functors

本章讲述泛化仿函数（generalized functors），这是一种威力强大的抽象概念，用以降低对象之间的联系（decoupled interobject communication）。泛化仿函数对那种“有必要将请求（requests）存储于对象内”的设计特别有用。设计模式 Command（Gamma 等,1995）用来描述“经过封装之请求”，泛化仿函数便是遵循这一模式。

简而言之，泛化仿函数是“将 C++ 所允许之任何处理请求（processing invocation）封装起来”后，获得的“具备型别安全性质”（typesafe）的高级对象。更详尽地说，泛化仿函数：

- 可封装任何处理请求（processing invocation），因为它可接受函数指针、成员函数指针、仿函数，甚至其他泛化仿函数——连同它们的某些引数或全部引数。
- 具备型别安全性（typesafe），因为它绝不会将错误的引数型别匹配到错误的函数上。
- 是一种带有“value 语义”的对象，因为它充分支持拷贝、赋值和传值（pass by value）。泛化仿函数允许被任意拷贝，并且不会暴露其虚成员函数。

藉由泛化仿函数，你可以将“处理请求”存储为数值，作为参数来传递，并在远离其创建点之处调用它们。它们是函数指针的更高级版本。函数指针和泛化仿函数之间的本质区别在于：后者可以存储状态（state），并可以调用成员函数。

阅读本章之后，你将能够：

- 理解 Command 设计模式，及其与泛化仿函数之间的联系。
- 知道 Command 模式和泛化仿函数何时有用。
- 了解 C++ 之中各种“函数型物体”（functional entities）的结构，以及如何在统一接口下封装它们。
- 知道如何在对象中存储一个处理请求（processing request）及其某些（或全部）参数，以及如何任意地传递它、调用它。
- 串联起多个“推迟的”调用动作，令它们依序执行。
- 知道如何使用功能强大的 Functor，那是一个实现了上述功能的 class template。

5.1 Command 设计模式

根据 Gamma 等四人 (GoF) 在其著作 (Gamma 等, 1995) 中所说, Command 模式的目的是为了“将请求封装于对象之内”。Command 对象是一个“和其实际执行者分开存储”的工件, 基本结构如图 5.1 所示。

这个模式的主要部件是 Command class 本身, 其最重要目的是降低系统中两个部分 (请求者 **invoker** 和接收者 **receiver**) 之间的依存性。

典型的行为次序应该像下面这样:

1. 应用程序 (客户端) 产生一个 **ConcreteCommand** 对象, 并传给它足够信息以备执行某项任务。图 5.1 中的虚线说明一个事实: 客户端会影响 **ConcreteCommand** 的状态。
2. 应用程序将 **ConcreteCommand** 对象中的 **Command** 接口传给 **invoker** (请求者), 由它保存这个接口。
3. 此后, 一旦调用者认为执行时机已到, 便启动 **Command** 的 **Execute()** 虚函数。虚拟调用机制会将这个调用动作发送给 **ConcreteCommand** 对象, 由后者处理细节。**ConcreteCommand** 找到 **Receiver** 对象 (任务执行者), 并通过该对象实际进行处理 (例如调用其 **Action()** 成员函数)。另一种情况是由 **ConcreteCommand** 对象全权处理, 此时图中的 **receiver** 不复存在。

invoker (请求者) 可以好整以暇地调用 **Execute()**。最重要的是执行期间你可以替换 **invoker** 所保存的 **Command** 对象, 从而将各式各样的行为安插到 **invoker** 对象中。

两件事情值得注意。第一, **invoker** 不知道工作是如何完成的。这不是什么新概念——使用排序算法时你也不必知道它是如何实作出来的。**Command** 的特别之处在于, **invoker** 甚至不知道 **Command** 对象将会做何种处理 (与此对比的是, 你当然预期得到排序算法的效果)。**invoker** 只有在某个条件成立时才会调用它所保存的 **Command** 接口中的 **Execute()**。另一方面, **receiver** 也不知道其 **Action()** 成员函数是否被 **invoker** 或其他什么对象调用。

因此, **Command** 对象确保了 **invoker** 和 **receiver** 之间的重要分离关系: 它们之间或许完全不互见, 但可通过 **Commands** 进行交流。通常 **invoker** 和 **receiver** 之间的连接线路由 **Application** 对象决定。这意味着对某一组 **receivers**, 你可以使用不同的 **invokers**; 还可以将不同的 **receivers** 挂接到某个特定的 **invoker** 身上——这一切都不需要它们彼此有任何了解。

第二, 让我们从时序 (timing) 的角度来观察 **Command** 模式。在一般编程任务中, 如果想执行一个动作, 你得将某个对象、该对象的某个成员函数、该函数的引数组装成一个调用。例如:

```
window.Resize(0, 0, 200, 100); // Resize the window
```

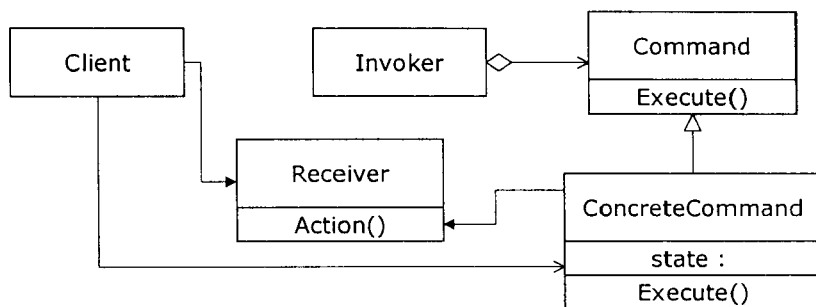


图 5.1 Command 设计模式

（续上页）“启动这样一个调用”的时刻和“收集这个调用所需元素（对象、函数、引数）”的时刻，在概念上是无法区分的。但在 Command 模式中，invoker 拥有调用动作的必要元素，却将调用动作无限延期。Command 模式就像下面例子一般地将调用延期：

```

Command resizeCmd(
    window,           // Object
    &Window::Resize,   // Member function
    0, 0, 200, 100);  // Arguments
// Later on...
resizeCmd.Execute(); // Resize the window
  
```

（C++ 的 `&Window::Resize` 较少为人所知，我将于稍后详加解释。）在 Command 模式中，收集“某处理动作所需环境（environment）”的时刻和执行该动作的时刻并不相同。在两个时刻之间，程序将该处理请求当做一个对象来保存和传递。如果没有这种时序（timing）上的需要，就不会有 Command 模式的存在。从这个角度看，Command 对象的存在归因于时序问题：由于你需要延后处理，所以你得有个对象将请求保存至那个时刻。

这些分析反映了 Command 模式的两个重要特点：

- 接口分离。invoker 和 receiver 分离。
- 时间分离。Command 保存了一个整装待发的处理请求，供将来运用。

环境（environment）概念也很重要。某执行点的“环境”指的是该执行点可见的一组实体（变量和函数）。当处理动作真正开始进行时，必要的环境必须准备就绪，否则处理动作无法执行开来。ConcreteCommand 对象可以将必要环境的一部分当作自身状态保存起来，并在执行 `Execute()` 期间取用其中部分消息。ConcreteCommand 保存的环境愈多，其独立性就愈强。

从实作角度来看，我们可以划分两类 concrete Command classes。一类仅仅将工作委托给 receiver。它们所做的事情只不过是对着一个 Receiver 对象调用其某个成员函数。我们称此为转发式命令 (forwarding commands)。另一类完成更复杂的工作，它们会调用其他对象的成员函数，也会嵌入一些单纯转发以外的逻辑。我们称此为主动式命令 (active commands)。

将命令划分为主动式和转发式，对于确立泛型实作范围 (scope) 十分重要。“主动式命令”无法加以灌装，它们的代码必然与应用程序有所关联，但我们却可以为“转发式命令”开发辅助工具。由于“转发式命令”的行为特征很像函数指针及其 C++ 兄弟：仿函数 (functor)，所以我们称之为泛化仿函数 (generalized functors)。

本章剩余篇幅的目标是设计一个 Functor class template，用以封装任何对象、该对象的任何成员函数，以及成员函数的任何引数。一旦被执行起来，Functor 将调动所有这些小部件，形成一个函数调用。

在一个用上 Command 模式的设计过程中，Functor 对象会带来极大帮助。在手工编写的实作码中，Command 模式的可伸缩性不够。你必须编写大量小型 concrete Command classes (分别对应于应用程序的每一个动作：CmdAdduser、CmdDeleteUser、CmdModifyUser...)，其中每个 class 都有一个平平淡淡的 Execute() 成员函数，用来转调用另一对象的某个成员函数。如果使用泛用型 Functor class，就可以将调用转发给任何对象的任何成员函数，对设计有很大帮助。

某些特殊的“主动式命令”也值得在 Functor 中实现，例如“多动作之序列化” (sequencing of multiple actions)。是的，Functor 应该有能力组装多个动作并依次执行它们。GoF 著作中称此种有用的对象为 MacroCommand。

5.2 真实世界中的 Command

关于 Command 模式，一个为人熟知的例子是窗口环境的设计问题。在良好的面向对象 GUI 框架中，Command 模式已经运用很多年了，只不过形式各异。

窗口系统的设计者需要一种泛化的 (通用的) 方法，将用户的操作 (例如鼠标点击、键盘敲击) 转达给应用程序。当用户点击按钮、选择功能表上的项目，或执行其他类似动作时，窗口系统必须将这些消息通知给相应的底部程序逻辑。从窗口系统的角度来看，[Tools] 选单下的 [Options] 命令不可包含任何特殊含义。如果它真有特殊含义，应用程序就会被绑死在一个非常死板的框架里头。想要将窗口系统和应用程序彼此分离，一个有效办法便是使用 Command 对象来传递用户动作。Commands 充当泛用型交通工具的角色，将用户动作从窗口传送至应用程序逻辑上。

在这个窗口实例中，invokers 是 UI (使用者接口) 相关元素 (例如按钮、检核框、选单、窗口部件)，receivers 是负责回应接口命令的对象 (例如某个对话框，或应用程序本身)。

Command 对象构成了 UI 和应用程序共同使用的“混合语言”。正如前一节所说, Command 提供了双重灵活性。首先,在不改变应用程序逻辑的情况下,你可以插入新式的 UI 元素。这就是所谓“可换肤 (skinnable)”概念,你可以在不改变产品本身设计的情况下添加“新皮肤”。表层皮肤不含任何架构 (architecture), 它们只提供“Commands 安插槽位”以及正确执行这些 Commands 的必要知识。其次,在不同的应用程序中,你可以轻易复用 (reuse) 相同的 UI 元素。

5.3 C++ 中的可调用体 (Callable Entities)

在着手“转发式命令” (forwarding commands) 的泛化实作之前,让我们先将“与命令相关”的概念和 C++ 编程中为人熟知的术语更具体地联系起来。

“转发式命令”实际上是一个超强的 callback (回调函数), 一个泛化的 callback。所谓 callback 是一个指针, 指向一个“可传递, 并随时可被调用”的函数。以下是一个示例:

```
void Foo();
void Bar();

int main()
{
    // Define a pointer to a function that takes no
    // parameters and returns void.
    // Initialize that pointer with the address of Foo
    void (*pF)() = &Foo;
    Foo();                // Call Foo directly
    Bar();                // Call Bar directly
    (*pF)();              // Call Foo via pF
    void (*pF2)() = pF;   // Create a copy of pF
    pF = &Bar;            // Change pF to point to Bar
    (*pF)();              // Now call Bar via pF
    (*pF2)();             // Call Foo via pF2
}
```

调用 Foo 和调用 (*pF) 有本质上的不同¹⁵: 后一种情况下你可以拷贝和修改函数指针。你可以取得一个函数指针将它保存于某处, 并于适当时候调用它——这和“转发式命令”类似, 本质上是一个“和其执行者分开存储, 并于日后被处理”的物体。

事实上 callback 是很多窗口系统运用 Command 模式时的一种 C 风格做法。例如 X Windows 在每个选单项和每个窗口部件 (widget) 中都保存这样一个 callback。当用户执行某项操作 (例如点击某个窗口部件) 时, 该部件会调用这个 callback。部件本身不知道 callback 实际上将做些什么。

除了简单的 callback, C++ 还定义了很多支持 function-call 操作符之物。下面列出 C++ 语言中支持 operator() 的所有构件:

¹⁵ 编译器提供了一种语法上的简化: (*pF)() 等同于 pF()。但 (*pF)() 更能反映实际发生的情况——首先是 pF 被提领 (dereference), 然后将 function call 操作符 (operator()) 施行于上。

- C-like function (C 风格的函数)。
- C-like pointer to function (C 风格的函数指针)。
- reference to function, 其行为本质上和 `const pointer to function` 类似。
- functor (仿函数), 亦即自行定义了 `operator()` 的一种对象。
- `operator.*` 或 `operator->*` 的施行结果。表达式左侧是一个 `pointer to member function`
- Constructor-call (构造函数)

对于上述所列的任何一项, 你可以在其右侧添加一对圆括号 `()`, 并在里头放入一组合适的参数, 用以执行某个处理动作。在 C++ 中, 除了以上所列, 再没有其他对象可以那么做。

支持 `operator()` 的对象被称为“可调用体 (callable entities)”。本章目标是实现一组“转发式命令”, 它们保存了一个调用并可该调用转发给任何一个可调用体¹⁶。Functor class template 将封装这些“转发式命令”, 并提供一致接口。

实作码必须处理三种基本情况: 一般函数调用, 仿函数调用 (包括对 Functor 的调用; 也就是说你应该可以将调用从某个 Functor 转发给另一个 Functor), 成员函数调用。你可能会想要定义一个抽象基类 (abstract base class) 并针对上述各种情况分别产生一个子类 (subclass)。这听起来像是很直截了当的 C++ 设计, 但是当你启动你喜爱的编辑器开始输入代码时, 一大堆问题便会冒出来。

5.4 Functor Class Template 骨干

说到 Functor 的实现, 我们当然要看看 handle-body 手法 (Coplien, 1992)。正如第 7 章详细讨论中所指出的那样, 在 C++ 中, 由于拥有权 (ownership) 方面的原因, 指向多态型别 (polymorphic type) 的普通指针并不严格具备高阶语义。为了解除 Functor 使用者对其生命周期的管理负担, Functor 最好具备“value 语义”, 也就是具备定义明确的拷贝和赋值操作。Functor 确实拥有一份多态实作, 但隐藏在内部, 我们称那个实作基类 (implementation base class) 为 FunctorImpl。 (译注: 关于 handle-body 手法, 亦可见《Effective C++》条款 34)

现在让我们做一次重要观察。Command 模式中的 `Command::Execute()` 应该成为 C++ 中的一个使用者自定义的 `operator()`。之所以在这儿使用 `operator()`, 有一个充足论点: 对 C++ 程序员来说, function-call 操作符具备“执行”或“做某事”的精确含义。另一个更有趣的论点是: 语法上的一致性。一个转发式 Functor 不但会委托 (delegates) 可调用体, 它本身也是一个可调用体。这就造成“一个 Functor 能够拥有其他 Functors”。所以从现在开始, Functor 构成了可调用体集合的一部分; 这使我们将来得以用更一致的方式来看待事物。

一旦开始定义 Functor wrapper (外覆类), 问题随之而来。我们的第一次定义可能是这样:

¹⁶请注意, 这里刻意不使用“型别” (type) 一词。我们可以简单地说: “凡支持 `operator()` 的型别都是可调用体”。但是和表面所见相反的是, C++ 中有些东西虽不具有某种型别, 你却可以对它施行 `operator()`。很快你会看到这一点。

```
class Functor
{
public:
    void operator()();
    // other member functions
private:
    // implementation goes here
};
```

第一个问题出在 `operator()` 的返回型别上。应该是 `void` 吗？某些情况下你可能会传回其他东西，例如 `bool` 或 `std::string` 之类。我们没有理由不将返回值参数化。

`template` 就是用来解决这类问题的，所以，二话不说：

```
template <typename ResultType>
class Functor {
public:
    ResultType operator()();
    // other member functions
private:
    // implementation
};
```

看上去还不错，我们现在拥有的不是“一个”`Functor`；我们拥有的是“一系列”`Functors`。这非常合情合理，因为传回字符串的 `Functors` 和传回整数的 `Functors` 在功能上是不同的。

现在看看第二个问题：难道 `Functor operator()` 不该接受引数吗？你可能想向这个 `Functor` 传递某些信息，而这些信息在 `Functor` 构造期间并不具备。例如，如果“鼠标在窗口上点击”的信息是通过 `Functor` 传递，那么调用这个 `Functor` 对象之 `operator()` 时，调用者应该将鼠标的位置传给它（调用时这些信息才确切存在）。

此外，在泛型世界中，参数的个数应该是任意的，参数的型别也应该是任意的。没有理由对此二者设限。

基于这些事实，我们的结论是：每一个 `Functor` 都根据返回型别和引数型别有所区分。这里需要的语言支持听起来有点可怕：数量可变的 `template` 参数，以及数量可变的函数参数。

遗憾的是这类语言支持非常缺乏。“数量可变的 `template` 参数”根本不存在。“数量可变的函数引数”虽然的确存在于 C++ 中，就像 C 那样，但你得小心，它们可以为 C 完成很出色的工作，却不能在 C++ 中有一样好的表现。“可变引数”系通过令人提心吊胆的省略符（像 `printf` 或 `scanf` 那种形式）来实现。调用 `printf()` 或 `scanf()` 时，传递的引数个数或型别如果不符合“指定格式”所载明的项目，是一种常见而危险的错误，同时也诠释了这类（使用省略符）函数的缺点。可变参数机制不安全、低阶、不适合 C++ 对象模型。让我长话短说：只要使用省略符，你就置身于无“型别安全”、无“对象语义”（对完整对象使用省略符，会导致不确定的行为）、无“`reference` 型别支持”的世界。甚至，被调用函数连自己获得的引数个数也无法得知。的确，哪儿出现省略符，哪儿就没剩多少 C++ 了。

另一种解法是，将 `Functor` 接受的引数个数限制为某个数量（合理大小）。在毫无根据的情况下进行选择，是程序员最不乐意做的事情之一。但毕竟我们至少有过去的经验可以依赖。程序

库（特别是老旧些的程序库）的函数所使用的参数通常最多不超过 12 个。现在让我们将引数个数限制在 15 个以下。就这样吧，把这个“任意个数”的问题抛一边去吧，别再为它烦恼。

即使做出了这样的决定，生活并没有好过多少。C++ 不允许同名而参数个数不同的 templates 存在，也就是说以下代码不合法：

```
// Functor with no arguments
template <typename ResultType>
class Functor
{
    ...
};

// Functor with one argument
template <typename ResultType, typename Parm1>
class Functor
{
    ...
};
```

如果为了解决问题而将它们命名为 `Functor1`、`Functor2`... 那会带来很大麻烦。

第 3 章曾经定义过 `typelists`，那是一种处理“型别集”的通用设施。`Functor` 的参数型别确实组成了一个型别集，所以 `typelists` 用在这里非常合适。这样的 `Functor` 长相大致如下：

```
// Functor with any number and types of arguments
template <typename ResultType, class TList>
class Functor
{
    ...
};
```

下面是可能的一份具现体 (instantiation)：

```
// Define a Functor that accepts an int and a double and
// returns a double
Functor<double, TYPELIST_2(int, double)> myFunctor;
```

这个解法有个显著优点：我们可以复用 `typelist` 设施带来的所有好处，不必再为 `Functor` 开发类似的东西。

很快你会看到，虽然 `typelists` 很有帮助，但 `Functor` 实作码还是需要做些辛苦的重复性工作，才能包含“任意个引数”。从现在起，让我们只考虑最多两个引数。头文件 `Functor.h` 将数目提高到 15，一如稍早所言。

被 `Functor` 包覆的那个多态类 `FunctorImpl` 具有和 `Functor` 一样的 `template` 参数¹⁷:

```
template <typename R, class TList>
class FunctorImpl;
```

`FunctorImpl` 定义出一个用以将“函数调用行为”抽象化的多态接口。针对不同的参数个数，我们分别定义相应的 `FunctorImpl` 显式特化版本 (explicit specialization, 见第 2 章)，其中针对自己的参数个数和型别，定义一个纯虚函数 `operator()`，如下所示：

```
template <typename R>
class FunctorImpl<R, NullType>
{
public:
    virtual R operator()() = 0;
    virtual FunctorImpl* Clone() const = 0;
    virtual ~FunctorImpl() {}
};
```

```
template <typename R, typename P1>
class FunctorImpl<R, TYPELIST_1(P1)>
{
public:
    virtual R operator()(P1) = 0;
    virtual FunctorImpl* Clone() const = 0;
    virtual ~FunctorImpl() {}
};
```

```
template <typename R, typename P1, typename P2>
class FunctorImpl<R, TYPELIST_2(P1, P2)>
{
    // 译注：再次提醒，讨论范围只限两个参数
public:
    virtual R operator()(P1, P2) = 0;
    virtual FunctorImpl* Clone() const = 0;
    virtual ~FunctorImpl() {}
};
```

这些 `FunctorImpl` classes 都是原始那个 `FunctorImpl` template 的偏特化版。第 2 章曾经就所谓“偏特化”做过详细论述，本例运用这项技术便可让我们定义出不同版本的 `FunctorImpl`——具体取决于 `typelist` 内的元素个数。

除了 `operator()`，`FunctorImpl` 还定义两个辅助函数——`Clone()` 和一个虚析构函数 (virtual destructor)。 `Clone()` 的目的是为了产生 `FunctorImpl` 对象的一份多态拷贝 (polymorphic copy, 请参阅第 8 章)，虚析构函数让我们得以在 `FunctorImpl` 指针上调用 `delete`，摧毁 `FunctorImpl` 的派生对象。第 4 章有详尽讨论，告诉你这个“什么也不做”的析构函数为什么很重要。

¹⁷ 不论使用关键字 `typename` 或 `class` 来表现 `template` 参数，都没有区别。本书习惯以 `typename` 表示那些可为基本型别 (如 `int`) 的参数，而以 `class` 表示那些必须是使用者自定义型别的参数。

Functor 遵循典型的 handle-body 实现手法:

```
template <typename R, class TList>
class Functor
{
public:
    Functor();
    Functor(const Functor&);
    Functor& operator=(const Functor&);
    explicit Functor(std::auto_ptr<Impl> spImpl);
    ...
private:
    // Handy type definition for the body type
    typedef FunctorImpl<R, TList> Impl;
    std::auto_ptr<Impl> spImpl_;
};
```

Functor 拥有一个指向 FunctorImpl<R, TList> (也就是其相应的 *body* 型别) 的智能指针 (smart pointer), 作为其 private 成员。这个智能指针用的是标准的 `std::auto_ptr`。

上述代码还演示了其他一些 Functor 部件: default 构造函数、copy 构造函数、assignment 操作符。这些部件的存在证实 Functor 具备 “*value* 语义”。这里不需要明白撰写析构函数, 因为 `auto_ptr` 会自动清除资源。

Functor 还定义了一个扩展 (extension) 构造函数, 接受一个 `auto_ptr` 指向 FunctorImpl。这个构造函数允许你 “定义 FunctorImpl 的派生类, 然后通过指向那些类的指针, 直接将 Functors 初始化”。为什么以 `auto_ptr` 为引数, 而不采用一般指针呢? 因为从外部看来, “根据 `auto_ptr` 完成构造” 可向外界明白透露出 “Functor 取得了 FunctorImpl 对象拥有权” 的消息。无论什么时候调用这个构造函数, Functor 使用者确实得敲下 `auto_ptr` 这几个字; 如果他们敲下 `auto_ptr` 这几个字, 他们应该知道 `auto_ptr` 是怎么回事¹⁸。

5.5 实现 “转发式” (Forwarding) Functor::operator()

Functor 需要一个可以转发至 FunctorImpl::operator() 的 operator()。我们可以使用曾经在 FunctorImpl 中使用过的相同手法: 针对不同的参数个数提供一组相应的 Functor 偏特化版本, 但是此法在这里不适用, 因此 Functor 定义了大量代码。如果仅仅因为 operator() 就要复制所有那些代码, 将造成很大的浪费。

¹⁸ 当然啦, 这不完全是事实, 但这总比无声无息地做一个选择 (这里是指 “拷贝” 和 “获取拥有权” 之间的选择) 要好。良好的 C++ 程序库具有这样一个有趣的特征: 任何地方只要可能造成歧义 (模棱两可), 它们都允许用户明确写出一些代码来消除歧义。当然也有些程序库误用隐式的 (silent) C++ 特性 (特别是 “型别转换” 和 “指针拥有权” 方面), 那样做可以让用户少写一些代码, 代价则是为用户做出了一些不确定的假设和决定。

首先，让我们定义参数的型别。Typelists 在这儿很有帮助：

```
template <typename R, class TList>
class Functor
{
    typedef TList ParmList;
    typedef typename TypeAtNonStrict<TList, 0, EmptyType>::Result
        Parm1;
    typedef typename TypeAtNonStrict<TList, 1, EmptyType>::Result
        Parm2;
    ... as above ...
};
```

`TypeAtNonStrict` 是个 `template`，用以取得 `typelist` 中某个位置上的型别。如果无所得，回传值（一个内部类 `TypeAtNonStrict<...>::Result`）将被核定为 `TypeAtNonStrict` 的第三个 `template` 引数。我们选择 `EmptyType` 作为第三引数，顾名思义，那是一个“在大括号之间未含任何代码”的 `class`（请参阅第 3 章关于 `TypeAtNonStrict` 的详细介绍，以及第 2 章关于 `EmptyType` 的讨论）。总而言之，`ParmN` 是 `typelist` 的第 `N` 个型别，但如果 `typelist` 元素个数少于 `N`，获得的结果将是 `EmptyType`。

为实作出 `operator()`，我们得仰赖一个有趣的技巧。我们可以针对任意数量的参数，在 `Functor` 定义式中定义出所有版本的 `operator()`，像下面这样：

```
template <typename R, class TList>
class Functor
{
    ... as above ...
public:
    R operator()() {
        return (*spImpl_)();
    }
    R operator()(Parm1 p1) {
        return (*spImpl_)(p1);
    }
    R operator()(Parm1 p1, Parm2 p2) {
        return (*spImpl_)(p1, p2);
    }
};
```

这里用到了什么技巧？对于某个给定的 `Functor` 具现体（`instantiation`），以上只有一个 `operator()` 是正确的，其他都存在编译期错误。你可以预期 `Functor` 完全没被编译出来。这是因为，每个 `FunctorImpl` 特化版本只定义了一个 `operator()`，而不是像 `Functor` 那样定义了一大群。这个技巧依赖一个事实：如果 `template` 的成员函数未曾被真正用上，C++ 不会将它具现化。因此，除非你错用 `operator()`，否则编译器不会报错。而如果你错用了某个 `operator()` 重载函数，编译器会尝试产出 `operator()` 函数，从而发现无法匹配。例如：

```
// Define a Functor that accepts an int and a double and
// returns a double.
Functor<double, TYPELIST_2(int, double)> myFunctor;
// Invoke it.
// operator()(double, int) is generated.
double result = myFunctor(4, 5.6);
// Wrong invocation.
double result = myFunctor();    // error!
// operator()() is invalid because
// FunctorImpl<double, TYPELIST_2(int, double)>
// does not define one.
```

由于有这样的美妙招数，我们不需要针对 0,1,2...个参数——对 `Functor` 进行偏特化，那将产生大量重复代码。我们只需定义所有 `operator()` 版本，再由编译器产生唯一被使用的那个。

现在东风俱备，我们可以开始定义从 `FunctorImpl` 派生出来的具象类 (concrete classes) 了。

5.6 处理仿函数

让我们从仿函数的处理开始。仿函数的定义很松散，任何 `class` 实体只要定义有 `operator()`，都是仿函数。`Functor` 满足这一定义，所以 `Functor` 是仿函数。因此，“以仿函数 `Fun` 之对象为参数”的 `Functor` 构造函数，是个“被 `Fun` 参数化”的 `templated` 构造函数：

```
template <typename R, class TList>
class Functor
{
    ... as above ...
public:
    template <class Fun>
    Functor(const Fun& fun);    // 译注：“以仿函数 Fun 之对象为参数”的 Functor 构造函数
};
```

为实作出这个构造函数，我们需要一个从 `FunctorImpl<R, TList>` 派生而来的 `class template FunctorHandler`，其中保存了一个型别为 `Fun` 的对象，并将 `operator()` 转发给该对象。我将使用前一节正确实现 `operator()` 的相同技巧。

为避免对 `FunctorHandler` 定义太多 `template` 参数，我让 `Functor` 具现体本身成为一个 `template` 参数。这个参数集合了其他所有参数，因为它供应内部 `typedef`。

```
template <class ParentFunctor, typename Fun>
class FunctorHandler
    : public FunctorImpl
{
    <
        typename ParentFunctor::ResultType,
        typename ParentFunctor::ParmList
    >
```



```

{
public:
    typedef typename ParentFunctor::ResultType ResultType;

    FunctorHandler(const Fun& fun) : fun_(fun) {}
    FunctorHandler* Clone() const
    { return new FunctorHandler(*this); }

    ResultType operator()()
    {
        return fun_();
    }
    ResultType operator()(typename ParentFunctor::Parm1 p1)
    {
        return fun_(p1);
    }
    ResultType operator()(typename ParentFunctor::Parm1 p1,
        typename ParentFunctor::Parm2 p2)
    {
        return fun_(p1, p2);
    }
private:
    Fun fun_;
};

```

`FunctorHandler` 看上去很像 `Functor`，都将请求（requests）转发给所保存的一个成员变量。两者的主要差异在于前者系通过实值（*value*）而非指针来保存仿函数。这是因为仿函数一般都得是“带有正规拷贝语义”的非多态型别（*nonpolymorphic types with regular copy semantics*）。

请注意某些地方对于内部型别 `ParentFunctor::ResultType`、`ParentFunctor::Parm1`、`ParentFunctor::Parm2` 的运用。`FunctorHandler` 实作了一个简单的构造函数：`Clone()` 和多个 `operator()` 版本（编译器会选择合适的一个）。如果你使用非法的 `operator()` 重载函数，必然会出现编译错误。

`FunctorHandler` 实作码并无特别之处，这并不意味着它不需要你好好思考。下一节你会看到这个小小的 `class template` 为我们带来多么强大的泛用性（*genericity*）。

有了 `FunctorHandler` 的声明，本节稍早声明的 `Functor templated` 构造函数就很容易实现。

```

template <typename R, class TList>
template <typename Fun>
Functor<R, TList>::Functor(const Fun& fun)
    : spImpl_(new FunctorHandler<Functor, Fun>(fun));
{
}

```

这里并无文字剪贴错误，一开始的两组 `template` 参数是必要的：第一组 `template <class R, class TList>` 用于 `class template Functor`，第二组 `template <typename Fun>` 则作为构造函数的参数。*C++ Standard* 将这类代码称为“位于 `class` 本体之外的 `member template` 定义式”。

构造函数的函数本体将对 `spImpl_` 成员进行初始化，使之指向一个型别为 `FunctorHandler` 的新对象，该对象系通过适当参数被具现化和初始化。

这里有几个地方值得一提，可以帮助我们正确理解建造中的 `Functor` 实作品。请注意，一旦进入 `Functor` 的这个构造函数，我们就可以通过 `template` 参数 `Fun` 了解全部的型别知识。一旦离开这个构造函数，在 `Functor` 运用范围内，型别信息就丢失了，因为 `Functor` 所知道的全部只有 `spImpl_`，它指向基类 `FunctorImpl`。这种“型别信息丢失现象”很有趣：构造函数知道型别，而且像工厂（factory）那样经由多态行为转换该型别。型别信息保存在 `FunctorImpl` 指针的动态型别中。

虽然我们还未写出多少代码（只写了一些“仅只一行”的函数），但已经可以测试一下了：

```
// Assume Functor.h includes the Functor implementation
#include "Functor.h"
#include <iostream>
// The using directive is okay in a small C++ program
using namespace std;

// Define a test functor
struct TestFunctor {
    void operator()(int i, double d) {
        cout << "TestFunctor::operator() (" << i
              << ", " << d << ") called.\n";
    }
};

int main() {
    TestFunctor f;
    Functor<void, TYPELIST_2(int, double)> cmd(f);
    cmd(4, 4.5);
}
```

这个小程序应该输出：

```
TestFunctor::operator()(4, 4.5) called.
```

这说明我们已经达到目标。现在可以进入下一步。

5.7 做一个，送一个

阅读前一节时，你可能会问自己：为什么不从“支持一般函数指针”开始呢？那应该是最简单的情况啊？为什么直接跳到仿函数（functor）、模板（template）…等等呢？答案很简单：走到这里，我们已经完成了对一般函数的支持。让我稍微修改测试程序：

```

#include "Functor.h"
#include <iostream>
using namespace std;

// Define a test function
void TestFunction(int i, double d) {
    cout << "TestFunction(" << i
        << ", " << d << ") called." << endl;
}

int main() {
    Functor<void, TYPELIST_2(int, double)> cmd(
        TestFunction);
    // will print: "TestFunction(4, 4.5) called."
    cmd(4, 4.5);
}

```

这份意外的惊喜要归功于 **template** 引数推导。当编译器看到需要从 `TestFunction` 构造出 `Functor` 时，它别无选择地执行前述的 **templated** 构造函数。于是编译器运用 **template** 参数 `void (&)(int, double)`，亦即 `TestFunction` 的型别，将此 **templated** 构造函数具现化。这个构造函数会具现出 `FunctorHandler<Functor<...>, void (&)(int, double)>`，造成 `FunctorHandler` 中的 `fun_` 的型别也是 `void (&)(int, double)`。当你调用 `FunctorHandler<...>::operator()` 时，调用会被转发至 `fun_()`——这是“通过函数指针调用函数”的合法语句。因此，`FunctorHandler` 基于两个事实自然而然地支持了函数指针：(1) 函数指针和仿函数在语法上的相似性，(2) C++ 的型别推导机制。

但是有一个问题（任何事情都不可能尽善尽美不是吗）。如果你重载 `TestFunction`（或重载传给 `Functor<...>` 的任何函数），你就得想办法消除某种歧义性。原因是如果 `TestFunction` 被重载，其名称（符号）所代表的型别就不再有明确定义。为说明这一点，让我在 `main` 之前为 `TestFunction` 增加一个重载版本：

```

// Declare an overloaded test function
// (no definition necessary)
void TestFunction(int);

```

突然之间猪羊变色，编译器抱怨说它无法确定该使用哪一个 `TestFunction` 重载版本，因为两个函数同名，该名称不再有识别功能。

就本质而言，如果出现重载，两种方法可以识别其中某个函数：一使用赋值（assignment），二使用转型（cast）。下面分别说明这两种方法：

```

// as above, TestFunction is overloaded
int main()
{
    // Typedef used for convenience
    typedef void (*TpFun)(int, double);
}

```

```

// Method 1: use an assignment
TpFun pF = TestFunction;
Functor<void, int, double> cmd1(pF);           // Ok
cmd1(4, 4.5);
// Method 2: use a cast
Functor<void, int, double> cmd2(
    static_cast<TpFun>(TestFunction));        // Ok
cmd2(4, 4.5);
}

```

赋值 (assignment) 和静态转型 (static cast) 都可以让编译器知道：你感兴趣的实际上是“参数为 int 和 double，返回型别为 void”的那个 TestFunction 函数。

5.8 引数和返回型别的转换

理想世界中，我们希望对 Functor 的转换能够像对一般函数调用动作的转换一样有效。也就是说，我们希望看到下面这样的情况：

```

#include <string>
#include <iostream>
#include "Functor.h"
using namespace std;

// Ignore arguments--not of interest
// in this example
const char* TestFunction(double, double)
{
    static const char buffer[] = "Hello, world!";
    // It's safe to return a pointer to a static buffer
    return buffer;
}

int main()
{
    Functor<string, TYPELIST_2(int, int)> cmd(TestFunction);
    // Should print "world!"
    cout << cmd(10, 10).substr(7);
}

```

虽然 main() 之中所使用的与实际的 TestFunction 标记式 (signature) 稍有不同 (后者接受两个 double，传回一个 const char*)，但还是应该能够绑定于 Functor<string, TYPELIST_2(int, int)> 身上。之所以会有这样的期望，因为 int 可以隐式转型为 double，const char* 可以隐式转型为 string。如果 Functor 不支持 C++ 的这种转型，就不成其为严格的 functor。

不必添加任何代码，我们就可以满足这一新要求。使用目前的成果，上面这个例子就可以顺利通过编译，并且如我们所期待地运作。为什么？答案仍旧在于我们对 FunctorHandler 的定义方式。

让我们看看，在 `template` 具现化过程尘埃落定之后，上述实例中的代码都做了些什么。函数：

```
string Functor<...>::operator()(int i, int j)
```

会转发（转调用）至虚函数：

```
string FunctorHandler<...>::operator()(int i, int j)
```

而这个虚函数的实作码最终会调用：

```
return fun_(i, j);
```

其中 `fun_` 的型别是 `const char* (*)(double, double)`，并被核定为 `TestFunction`。

当编译器遇到对 `fun_` 的调用动作时，它会正常编译，就像编译你手写的码一样。这里所谓“正常”是指“转型规则的运用”和平常一样。然后，编译器产生代码，将 `i, j` 转型为 `double`，将返回型别转型为 `std::string`。

`FunctorHandler` 的泛用性和灵活性展示了“代码自动生成”（code generation）的强大威力。在泛型程序设计中，`template` 引数对其相应参数的语法替换是很典型的情况。`template` 的处理发生在编译之前，这让你得以在源码层级有所动作。面向对象程序设计与此并无不同，它的威力来自于“名称（name）”和“实值（value）”的后期（编译后）绑定。因此，面向对象编程支持二进制组件形式的复用，泛型编程支持源码层级的复用。与二进制码相较，源码天生具有更多信息和更高级别，所以泛型编程支持更丰富的构件，但其代价是较弱的执行期动态性。你无法通过 STL 完成 CORBA 所能完成的功能，反之亦然。两种技术相互补充。

现在，藉由同一个程序库，我们可以处理所有种类的仿函数，以及一般函数。另一个收获是，引数和返回值型别也具备隐式转型能力。

5.9 处理 Pointer to member function (成员函数指针)

译注：本节讨论 `operator.*` 和 `operator.->`。关于这两个操作符，Scott Meyers 发表过一篇非常好的文章，解说十分详细：

"Implementing operator->* for smart pointers", Scott Meyers, DDJ, Oct., 1999

虽然成员函数指针（pointers to member functions）在日常编程工作中不很常用，但有时候它还会很有帮助。它们的行为很像函数指针，但当你想调用它们时，除了引数之外，你还必须传递一个对象。以下例子示范成员函数指针的语法和语义。

```
#include <iostream>
using namespace std;

class Parrot
{
public:
    void Eat() {
        cout << "Tsk, knick, tsk...\n";
    }
}
```

```

void Speak() {
    cout << "Oh Captain, my Captain!\n";
}
};

int main() {
    // Define a type: pointer to a member function of
    // Parrot, taking no arguments and
    // returning void.
    typedef void (Parrot::* TpMemFun)();

    // Create an object of that type
    // and initialize it with the address of
    // Parrot::Eat.
    TpMemFun pActivity = &Parrot::eat;
    // Create a Parrot...
    Parrot geronimo;
    // ...and a pointer to it
    Parrot* pGeronimo = &geronimo;

    // Invoke the member function stored in Activity
    // via an object. Notice the use of operator.*
    (geronimo.*pActivity)();

    // Same, via pointer. Now we use operator->*
    (pGeronimo->*pActivity)();
    // Change the activity
    pActivity = &Parrot::Speak;

    // Wake up, Geronimo!
    (geronimo.*pActivity)();
}

```

更仔细地看看成员函数指针和与之相关的两个操作符 `.*` 和 `->*`，我们会发现一些奇怪的特点。`geronimo.*pActivity` 和 `pGeronimo->*pActivity` 的运算结果并没有对应的 C++ 型别。二者的确都是一个二元操作，并且都会传回某个东西，让你可以立即对它施行 function-call 操作符，但这里所谓的“某个东西”不具型别¹⁹。你无法通过某种方式将 `operator.*` 或 `operator->*` 的操作结果保存下来，虽然的确有某个物体保存着你的对象和成员函数指针间的连接点。这个连接点似乎非常不稳定，只在你调用 `operator.*` 或 `operator->*` 时它才会从混沌世界中现身，并存在“恰好让你足以调用 `operator()`”的时间，然后又返回混沌世界。你无法对它进行任何其他处理。

¹⁹ C++ Standard 说：如果 `.*` 或 `->*` 的操作结果是个函数，这个结果就只能被当做 function-call 操作符 `operator()` 的操作数 (operand)。

在 C++ 中，每个对象都有型别，但 `operator->*` 和 `operator.*` 的运行结果是唯一的例外。这比“重载所导致的函数指针歧义性”（前一节作了介绍）更棘手，彼时我们得从太多型别中做出抉择，但我们可以消除选择所导致的歧义。此刻我们却没有任何型别可以着手。因此，成员函数指针和与其相关的两个操作符是 C++ 中一个“半生不熟”的概念。顺便说一句，虽然你可以获得一般函数的 `reference`，却无法获得成员函数的 `reference`。

有些 C++ 编译器厂商定义出一种新型别，让你可以通过以下语法保存 `operator.*` 操作结果：

```
// __closure is a language extension      // 译注：延续上例脉络
// defined by some vendors
void (__closure:: * geronimosWork)() =
    geronimo.*pActivity;
// Invoke whatever Geronimo is doomed to do
geronimosWork();
```

`geronimosWork` 的型别中并没有 `Parrot` 的任何信息。这意味着日后你可以将 `geronimosWork` 绑定至 `geronimo` 以外的某个东西身上，甚至绑定至非属 `Parrot` 的某个东西上。你需要知道的仅仅只是成员函数指针的返回型别和引数。事实上这个“语言扩充性质”是一种 `Functor class`，但只限于处理对象和成员函数（不允许处理一般函数或仿函数）。

现在让我们开始为 `Functor` 提供“成员函数指针”的绑定支持。有了仿函数和一般函数的经验，我们知道，“保持事物的泛用性”而不要“过早跳进特定性的怪圈内”绝对是一件好事。在 `MemFunHandler` 实作码中，对象型别（前例的 `Parrot`）是个 `template` 参数。此外，我们也让成员函数指针成为一个 `template` 参数。这样一来，我们便免费获得了自动转型功能，就像 `FunctorHandler` 实作码的情形一样。

下面便是 `MemFunHandler` 的实作码，具体实现了以上讨论的想法，以及 `FunctorHandler` 已采用的一些想法。

```
template <class ParentFunctor, typename PointerToObj,
         typename PointerToMemFn>
class MemFunHandler
    : public FunctorImpl
{
    <
        typename ParentFunctor::ResultType,
        typename ParentFunctor::ParmList
    >
public:
    typedef typename ParentFunctor::ResultType ResultType;
    MemFunHandler(const PointerToObj& pObj, PointerToMemFn pMemFn)
        : pObj_(pObj), pMemFn_(pMemFn) {}

    MemFunHandler* Clone() const
    { return new MemFunHandler(*this); }
```

```

ResultType operator()() {
    return ((*pObj_).*pMemFn_){};
}

ResultType operator()(typename ParentFunctor::Parm1 p1) {
    return ((*pObj_).*pMemFn_)(p1);
}

ResultType operator()(typename ParentFunctor::Parm1 p1,
    typename ParentFunctor::Parm2 p2) {
    return ((*pObj_).*pMemFn_)(p1, p2);
}

private:
    PointerToObj pObj_;
    PointerToMemFn pMemFn_;
};

```

为什么将 `MemFunHandler` 以指针型别 (`PointerToObj`) 为参数, 而不以对象本身的型别为参数呢? 更直截了当的写法应该像这样:

```

template <class ParentFunctor, typename Obj,
    typename PointerToMemFn>
class MemFunHandler
    : public FunctorImpl
    {
    public:
        MemFunHandler(Obj* pObj, PointerToMemFn pMemFn)
            : pObj_(pObj), pMemFn_(pMemFn) {}
        ...
    };

```

这段程序似乎更容易让人理解, 但是前述第一个实作版本较具泛用性。第二份实作把对象指针的型别写死在其实作码内, 它存储的是一个指向 `Obj` 的“赤裸裸、光秃秃、未加修饰的”指针。这有问题吗?

是的, 有问题——如果你想对 `MemFunHandler` 使用 `smart pointer` (智能指针) 的话。啊哈! 第一份实作版本支持 `smart pointer`, 第二份不行。第一份实作版本能够保存“行为像指针”的任何型别; 第二份则仅能为一般指针服务, 而且第二份版本不支持“指向 `const` 对象”的指针。这就是将型别写死于代码所带来的负作用。

让我们针对新实现的功能就地进行一次测试。现在 `Parrot` 被复用了 (`reused`)。


```

#include "Functor.h"
#include <iostream>
using namespace std;

class Parrot {
public:
    void Eat() {
        cout << "Tsk, knick, tsk...\n";
    }
    void Speak() {
        cout << "Oh Captain, my Captain!\n";
    }
};

int main() {
    Parrot geronimo;
    // Define two Commands
    Functor<>
        cmd1(&geronimo, &Parrot::Eat),
        cmd2(&geronimo, &Parrot::Speak);
    // Invoke each of them
    cmd1();
    cmd2();
}

```

由于 MemFunHandler 一开始就追求最大程度的泛用性，所以自动转型的好处得以免费获得——就像 FunctorHandler 的情况一样。

5.10 绑定 (Binding)

译注：此处绑定不是指 function call 和 function body 之间的绑定，而是对某物实值的“绑定”。比较接近 STL 中的 binder (binder1st, binder2nd)

我们本可就此止步。如今我们已经万事具备，Functor 可以支持先前讨论所定义的任何一种 C++ 可调用体 (callable entities)，而且表现得很出色。然而正如 Pygmalion (译注：茶花女，英国剧作家萧伯纳笔下人物) 指出：有时候，在你动手之前，你不可能预测实际成果。

Functor 的实现刚刚完成，新的想法又浮上我的心头。我们可能想将某种型别的 Functor 转换为另一种。绑定 (binding) 就是这样一种转换：假设有个 Functor 取两个整数作为参数，你想将其中一个整数绑定为某固定值，只让另一个变化。绑定会产出一个“只取单一整数”的 Functor，因为另一个是固定的，因而也是可知的。如下：

```

void f()
{
    // Define a Functor of two arguments
    Functor<void, TYPELIST_2(int, int)> cmd1(something);
    // Bind the first argument to 10
    Functor<void, TYPELIST_1(int)> cmd2(BindFirst(cmd1, 10));
}

```

```

    // Same as cmd1(10, 20)
    cmd2(20);
    // Further bind the first (and only) argument
    // of cmd2 to 30
    Functor<void> cmd3(BindFirst(cmd2, 30));
    // Same as cmd1(10, 30)
    cmd3();
}

```

绑定是一项威力强大的功能。你不但可以保存“可调用体 (callable entities)”，还可以保存它们的部分（或全部）引数。这大大提高了 **Functor** 的表达能力，因为它可以让你包装函数和引数，无需添加作为“粘胶”之用的代码。

假设你想在文字编辑器中实现 “Redo” 功能，当用户输入一个字符 ‘a’ 时，你调用成员函数 `Document::InsertChar('a')`，然后添加一个包装好的 **Functor**，其中内含指向 **Document** 的指针、成员函数 `InsertChar()` 以及实际字符。当用户选择 Redo 功能时，你只需启动那个 **Functor** 就万事大吉。5.14 节将就 **undo** 和 **redo** 功能作更进一步的讨论。

从一个更广泛的角度观之，绑定一样功能强大。让我们将 **Functor** 视为一个计算 (computation)，将它的引数看做是执行该计算所需的环境 (environment)。截至目前，**Functor** 可以藉由保存“函数指针”和“成员函数指针”的方式将计算过程延后，但是 **Functor** 所保存的只是计算，并没有保存和计算相关的环境。“绑定”可以让 **Functor** 将部分环境连同计算一起保存下来，并逐步降低调用时刻所需的环境需求。

具体实作之前，让我们整理一下需求。对于一个 **Functor<R, TList>** 具现体，我们想将第一引数 (`TList::Head`) 绑定为一个固定值。因此，返回型别将是 **Functor<R, TList::Tail>**。

有了这些，实现 **BinderFirst** class template 易如反掌。需要特别注意的地方只有一个：这里涉及两个 **Functor** 具现体，*incoming* **Functor** 和 *outgoing* **Functor**。前者的型别被当做参数 **ParentFunctor** 传递，后者的型别则用于计算。

```

template <class Incoming>
class BinderFirst
    : public FunctorImpl<typename Incoming::ResultType,
                       typename Incoming::Arguments::Tail>
{
    typedef Functor<typename Incoming::ResultType,
                  Incoming::Arguments::Tail> Outgoing;
    typedef typename Incoming::Param1 Bound;
    typedef typename Incoming::ResultType ResultType;

public:
    BinderFirst(const Incoming& fun, Bound bound)
        : fun_(fun), bound_(bound)
    {
    }

    DEFINE_CLONE_FUNCTORIMPL(BinderFirst)

```

```

ResultType operator() ()
{
    return fun_(bound_);
}
ResultType operator() (typename Outgoing::Parm1 p1)
{
    return fun_(bound_, p1);
}
ResultType operator() (typename Outgoing::Parm1 p1,
    typename Outgoing::Parm2 p2)
{
    return fun_(bound_, p1, p2);
}
private:
    Incoming fun_;
    Bound bound_;
};

```

`class template` `BinderFirst` 和以下的 `template function` `BindFirst` 协同工作。后者的优点在于，它可以根据你传入的实际引数的型别，自动推导出 `template` 参数。（译注：这和 STL 的 `Binder1st` 和 `Bind1st()` 的合作模式完全一样）

```

// See Functor.h for the definition of BinderFirstTraits
template <class Fctor>
typename Private::BinderFirstTraits<Fctor>::BoundFunctorType
BindFirst(
    const Fctor& fun,
    typename Fctor::Param1 bound)
{
    typedef typename
        private::BinderFirstTraits<Fctor>::BoundFunctorType
        Outgoing;
    return Outgoing(std::auto_ptr<typename Outgoing::Impl>(
        new BinderFirst<Fctor>(fun, bound)));
}

```

绑定和自动转型可谓天作之合，它为 `Functor` 带来了惊人的灵活性。下面的例子将绑定和自动转型结合起来运用：

```

const char* Fun(int i, int j)
{
    cout << Fun(" << i << ", " << j << ") called\n";
    return "0";
}

int main() {
    Functor<const char*, TYPELIST_2(char, int)> f1(Fun);
    Functor<std::string, TYPELIST_1(double)> f2(
        BindFirst(f1, 10));
    // Prints: Fun(10, 15) called
    f2(15);
}

```

5.11 将请求串接起来 (Chaining Requests)

GoF 著作 (Gamma 等, 1995) 给出了一个 `MacroCommand` class 实例, 这是一个命令, 保存着由 `Command` 构成的线性集合 (例如 `list` 或 `vector`)。当 `MacroCommand` 被执行起来时, 它会依序执行它所保存的每一道命令。

这个性质非常有用。举个例子, 让我们再说说先前那个 `undo/redo` 例子。一个 “do” 操作可能伴随着多个 “undo” 操作, 例如插入一个字符可能会使文本窗口自动卷动 (某些编辑器提供有此功能, 以保证更好的文本显示效果)。一旦执行 `undo` 操作, 你应该会希望窗口卷回原位置 (大多数编辑器无法正确恢复原位置, 真是讨厌)。为了回复至窗口卷动前的状态, 你需要在单一 `Functor` 对象中保存多个 `Commands`, 并将它们视为一个整体来执行。`Document::InsertChar()` 成员函数会将 `MacroCommand` 压入所谓的 `undo stack`, `MacroCommand` 则由 `Document::DeleteChar()` 和 `Window::Scroll()` 组成。后一个成员函数将绑定某个引数, 该引数保存着原始位置 (你看, 绑定功能带来如此便利)。

Loki 定义了一个 `FunctorChain` class 和一个辅助函数 `Chain()`。`Chain()` 声明如下:

```
template <class Fun1, class Fun2>
Fun2 Chain(
    const Fun1& fun1,
    const Fun2& fun2);
```

至于 `FunctorChain` 的实作则平淡无奇: 它保存两个仿函数, 并由 `FunctorChain::operator()` 依次调用它们。你可以多次调用 `Chain()`, 串联多个仿函数。

让我们以 `Chain()` 作为 `MacroCommand` 的讨论终点。在大量 “难能可贵” 的功能中, 有这样一点: `list` 永远可以增长。可供增长的空间很多。不论 `BindFirst` 或 `Chain()` 都不会导致对 `Functor` 的任何修改, 这就保证你自己也能够建造出类似设施。

5.12 现实世界中的问题之 1: 转发函数的成本

`Functor` class template 的总体设计已经完成。现在让我们集中心力优化它, 使它尽量高效²⁰。

让我们重点看看 `Functor` 的一个 `operator()` 重载函数——它把调用转发给 `smart pointer`:

```
// inside Functor<R, TList>
R operator()(Parm1 p1, Parm2 p2) {
    return (*spImpl_)(p1, p2);
}
```

²⁰ 一般而言 “过早优化” 是不被提倡的。原因之一是程序员不善于评估程序哪一部分应该优化, (更重要的是) 哪一部分不应该被优化。但程序库设计者的情况有所不同, 他们不知道程序库是否会被客户端程序的某个关键部分采用, 所以他们应该竭尽全力进行优化。

每次调用 `operator()`，这个函数都对每一个引数执行一次非必要的拷贝。如果 `Parm1` 和 `Parm2` 的拷贝成本高昂，就会带来效率上的问题。

奇怪的是，即使 `Functor` 的 `operator()` 是个 `inline` 函数，编译器也不会藉由优化消除这个额外的拷贝。Sutter (2000) 条款 46 曾就 C++ *Standard* 定案前做出的这个“语言上的最新变化”进行介绍。所谓“copy 构造的省略”对转发函数来说是不允许的。编译器省略 copy 构造函数的唯一情况只能是针对返回值，而这种优化无法手工进行。

reference 是不是可以轻松解决这个问题呢？让我们这样试试：

```
// inside Functor<R, TList>
R operator()(Parm1& p1, Parm2& p2)
{
    return (*spImpl_)(p1, p2);
}
```

看上去还不错，而且可能真的有效，除非你这么做：

```
void testFunction(std::string&, int);
Functor<void, TYPELIST_2(std::string&, int)> cmd(testFunction);
...
string s;
cmd(s, 5); //error!
```

编译器会在最后一行遇到麻烦，从而向你发出“reference to reference 是不允许的”这类消息（实际消息可能更含糊）。这里所发生的情况是，如此一个具现动作会使 `Param1` 成为 `std::string` 的 reference，于是 `p1` 成为 `std::string` 的 reference to reference，而那是非法的²¹。

幸运的是第 2 章刚好提供了一个解决这类问题的工具。那儿提供了一个 `class template` `TypeTraits<T>`，它定义一组“和型别 `T` 相关”的型别，实例包括 `non-const` 型别（如果 `T` 是 `const` 型别）、指针所指型别（如果 `T` 是指针）以及其他很多型别。可以“安全而高效地被当做函数参数传递”的型别是 `ParameterType`。下表列出“传给 `TypeTraits` 的型别”和“内部型别 `ParameterType`”之间的联系。请将 `U` 视为一般型别，例如某个 `class` 或基本型别。

T	TypeTraits<T>::ParameterType
U	如果 U 为基本型别，则为 U；否则为 const U &
const U	如果 U 为基本型别，则为 U；否则为 const U &
U &	U &
const U &	const U &

²¹ 这个问题也出现在 C++ *Standard* 制定者当中。Bjarne Stroustrup 已经给标准委员会提交了一份错误修改报告。他对这个问题的建议是：允许使用 reference to reference，并简单地将它视为 reference 来处理。本书撰写之际，这份报告可以在以下网址找到：

http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_active.html#106。

如果以右侧栏位中的型别替换转发函数的引数，任何情况下它都会正确工作，而且不需负担任何拷贝额外开销：

```
// Inside Functor<R, TList>
R operator() (
    typename TypeTraits<Parm1>::ParameterType p1,
    typename TypeTraits<Parm1>::ParameterType p2)
{
    return (*spImpl_)(p1, p2);
}
```

更棒的是，`reference` 可以顺利地跟 `inline` 函数协同工作——优化器更容易生成优化代码，因为它所做的只不过是跟 `reference` 进行简化。

5.13 现实世界中的问题之 2：Heap 分配

接下来让我们关注 `Functor` 的构造和拷贝成本。我们已经实现了正确的拷贝语义，但得付出“heap 分配”带来的成本。每个 `Functor` 都保存着一个 `smart pointer`，指向 `new` 配得的对象。拷贝一个 `Functor` 时，会伴随一个通过 `FunctorImpl::Clone` 执行的深拷贝 (`deep copy`)。

如果再将你所使用的对象大小考虑进来，事情就会变得特别麻烦。通常 `Functor` 会和“函数指针”、“对象指针和成员函数指针的组合 (一个 `pairs`)”一起使用。在典型 32 位系统中，这些对象分别占用 4bytes 和 20bytes (4 用于对象指针，16 用于成员函数指针²²)。绑定发生时，具象 (`concrete`) 仿函数对象的大小大致随着被绑定的引数的大小而增加。如果发生空额填补 (`padding`)，实际大小还会再多一些。

第 4 章引入一种高效的小型对象分配器。`FunctorImpl` 及其派生对象正适合运用那种分配器。请回忆一下第 4 章，小型对象分配器的使用方式之一是：让你的 `class` 派生自 `SmallObject class template`。

`SmallObject` 的用法非常简单，但我们需要为 `Functor` 和 `FunctorImpl` 增加一个 `template` 参数，以反映内存分配器所使用的线程模型 (`threading model`)。这不会带来麻烦，因为大多数情况下你会使用缺省引数。以下粗体字表示新修改的地方：

```
template
<
    class R,
    class TL,
    template <class T>
        class ThreadingModel = DEFAULT_THREADING,
>
```

²² 你可能会以为成员函数指针占用 4bytes，就像函数指针那样。但是成员函数指针实际上是带标记的 (`tagged`) unions，它们可以对付多重虚继承以及虚拟/非虚函数。

```
class FunctorImpl : public SmallObject<ThreadingModel>
{
public:
    ... as above ...
};
```

只需要对 `FunctorImpl` 做这些事，就可以完全运用第 4 章这个定制型分配器的好处。

同样道理，`Functor` 也得增加一个 `template` 参数：

```
template
<
    class R,
    class TL,
    template <class T>
        class ThreadingModel = DEFAULT_THREADING,
>
class FunctorImpl
{
    ...
private:
    // Pass ThreadingModel to FunctorImpl
    std::auto_ptr<FunctorImpl<R, TL, ThreadingModel> pImpl_;
```

现在如果你想经由缺省的线程模型（threading model）使用 `FunctorImpl`，并不需要指定第三引数。只有当程序需要“支持两个或多个线程”的 `Functors` 时，你才需要明确指定 `ThreadingModel`。

5.14 通过 Functor 实现 Undo 和 Redo

GoF 著作（Gamma 等, 1995）中建议将 `undo` 作为 `Command class` 的一个成员函数 `Unexecute` 来实现。问题是你无法用某种泛化方式来表达 `Unexecute`，因为“做某事”和“撤销某事”之间的关系无法预知。如果应用程序中的每个操作都有一个具象的 `Command class`，`Unexecute` 方案的确很吸引人；但 `Functor-based` 解法比较倾向使用单一 `class`，并让后者绑定不同的对象和成员函数调用。

Al Stevens 为 *Dr. Dobbs's Journal*（Stevens 1998）写过一篇文章，对于研究 `undo` 和 `redo` 的泛化实现很有帮助。他设计了一个 `undo/redo` 泛型库，在你着手自己的设计之前，应该看看他的作品，无论你是否使用 `Functor`。

这完全是数据结构方面的问题。`undo` 和 `redo` 的基本思想是，你必须维护一个 `undo stack` 和一个 `redo stack`。当用户“做”某些事，例如输入一个字符时，你就将一个“不同的”`Functor` 压入 `undo stack`。这意味着成员函数 `Document::InsertChar` 有责任将“正确的撤销动作”（例如 `Document::DeleteChar`）压入 `undo stack` 中。这会增加“承事者”成员函数的负担，`Functor` 则无需知道如何撤销自己。

此外，你可能会想要将一个 Functor 压入 redo stack。这个 Functor 由一个 Document 和一个 `Document::InsertChar()` 成员函数指针组成，而且都绑定至实际字符型别。某些编辑器允许“重复输入 (retyping)”：在你输入某些东西并选择 Redo 后，输入区块会被再次重复。我们为 Functor 设计的绑定功能极有助于保存“针对某一给定字符”的 `Document::InsertChar()` 调用，它将一切都封装在一个单独的 Functor 中。此外，不仅最后一个输入字符需要重复——那不是什么大不了的功能——而且最近一次“非字符输入行为”后的整串输入都应该重复。此时需要串联 (chaining) Functors：只要用户进行输入，你就将其添加至同一个 Functor 中。采用这种方法，你就可以将多次键盘输入当做单一动作来处理。

`Document::InsertChar` 会将 Functor 实际压入 undo stack。当用户选择 Undo 时，那个 Functor 会被执行并被压入 redo stack。

如你所见，“引数绑定”和“组合”技术可以让我们以十分一致的方式来处理 Functors：无论何种调用最终都会被包装在一个 Functor 中，这大大减轻了实现 undo 和 redo 所需的工作量。

5.15 摘要

在 C++ 中，使用一个好程序库比（或至少应该比）设计一个好程序库简单得多。然而从另一个角度看，设计程序库大有乐趣。回头看看前面所有实作细节，我们知道，撰写泛型代码时，有一些经验需要牢记在心：

- 只要涉及型别，请运用“模板化 (templating)”和“推迟”技巧。力求泛化 (通用性)。藉由 `template`，`FunctorHandler` 和 `MemFunHandler` 从“推迟型别信息”上获得了大量好处，轻易就得到了对函数指针的支持。与所得功能 (利益) 相比，代码极为小巧。这一切都得益于“`template` 的使用”以及“尽可能让编译器推导型别”。
- 鼓励高级语义。如果单单只和 `FunctorImpl` 指针合作，会令你大感头疼。想象一下你该怎样自行实现绑定 (binding) 和串联 (chaining)。

技术的运用是为了得到“简单化”的好处。讨论过模板 (template)、继承、绑定、内存之后，我们提炼出一个简单、易用、清晰明确的程序库。

简而言之，Functor 是对函数、仿函数或成员函数的推迟调用，它将被调用者保存下来并将 `operator()` 开放出来以供调用。让我们再次强调它的一些重点。

5.16 Functor 要点概览

- Functor 是一个“可表示调用动作 (并携带多达 15 个引数)”的 `template`。第一参数是调用函数的返回型别，第二参数是一个“内含调用函数之各个参数型别”的 `typelist`。第三参数用以建立 Functor 所使用之内存分配器的线程模型。`typelists` 细节请参阅第 3 章，线程模型的详细介绍请见本书附录，小型对象分配技术请参阅第 4 章。

- 你可以以一个函数、仿函数、Functor、对象指针、成员函数指针，将某个 Functor 初始化，例如：

```
void Function(int);

struct SomeFunctor {
    void operator()(int);
};

struct SomeClass {
    void MemberFunction(int);
};

void example()
{
    // Initialize with a function
    Functor<void, TYPELIST_1(int)> cmd1(Function);
    // Initialize with a functor
    SomeFunctor fn;
    Functor<void, TYPELIST_1(int)> cmd2(fn);
    // Initialize with a pointer to object
    // and a pointer to member function
    SomeClass myObject;
    Functor<void, TYPELIST_1(int)> cmd3(&myObject,
        &SomeClass::MemberFunction);
    // Initialize a Functor with another
    // (copying)
    Functor<void, TYPELIST_1(int)> cmd4(cmd3);
}
```

- 你也可以以 `std::auto_ptr< FunctorImpl<R, TList> >` 将 Functor 初始化。这使用户得以自行定义扩展部分。
- Functor 支持引数和返回值自动(型别)转换。例如先前的例子中，`Function`、`SomeFunctor::operator()`、`SomeClass::MemberFunction` 的引数可以是 `double` 而非 `int`。
- 涉及重载 (overloading) 时，需手动消除歧义 (模棱两可，ambiguation)。
- Functor 完全支持高级语义：拷贝 (copy)、赋值 (assignment)、传值 (pass by value)。Functor 不具多态性，不能派生 classes。如果想扩展 Functor，请从 `FunctorImpl` 派生。
- Functor 支持引数绑定。`BindFirst` 会将第一个引数绑定为某固定值，传回一个“以第一引数之外的其他引数”参数化 (parameterized) 的 Functor。例如：

```
void f()
{
    // Define a Functor of three arguments
    Functor<void, TYPELIST_3(int, int, double)> cmd1(
        someEntity);
}
```

```
// Bind the first argument to 10
Functor<void, typelist_2(int, double)> cmd2(
    BindFirst(cmd1, 10));
// Same as cmd1(10, 20, 5.6)
cmd2(20, 5.6);
}
```

- 使用 Chain 函数, 可将多个 Functors 串联在一个单独的 Functor 对象中, 例如:

```
void f()
{
    Functor<> cmd1(something);
    Functor<> cmd2(somethingElse);
    // Chain cmd1 and cmd2
    // as the container
    Functor<> cmd3(Chain(cmd1, cmd2));
    // Equivalent to cmd1(); cmd2();
    cmd3();
}
```

- Functor 的成本是, 即使只是简单的 Functor 对象也具有间接性 (通过指针进行调用)。每一个绑定都需付出一个额外的虚调用成本。串联 (chaining) 也需付出一个额外的虚调用成本。参数倒是不会被拷贝, 除非需要转换。
- FunctorImpl 使用第 4 章定义的小型对象分配器。

6

Singletons (单件) 实作技术

设计模式 Singleton (Gamma 等, 1995) 的独特之处在于, 它是一个奇怪的结合体: 描述十分简单, 实作却很复杂。书籍和刊物文章中大量与 Singleton 相关的讨论和实作码证实了这一点 (例如 Vlissides 1996, 1998)。GoF 著作中对 Singleton 的描述只是这么简单: “保证一个 class 只有一个实体 (instance), 并为它提供一个全局访问点 (global access point)”。

Singleton 是一种经过改进的全局变量。它所带来的改进是, 你无法产生第二个具有 Singleton 形态的对象。因而, 当你模塑 (model) 那些“从概念上来说, 在程序中只 (能) 有唯一实体”的型别, 例如键盘、显示器、打印管理器 (PrintManager)、系统时钟 (SystemClock) 时, 就应该使用 Singleton。如果这些型别能够被具现化为一个以上的实体, 那么你能期望的最好结果是“违反常理”, 更多情况下则会带来危险。

“提供一个全局访问点”具有微妙含义: 从客户角度来看, Singleton 对象“拥有自己”。客户产生 Singleton 时并不需要什么特殊步骤。Singleton 对象负责自身的诞生和摧毁。Singleton 生命期 (lifetime) 的管理是实作 Singleton 时最伤脑筋的地方。

对于 C++ 中各种 Singleton 变体的设计和实作, 本章讨论了与之相关的一些最重要主题:

- 与一般单纯的全局对象相比, Singleton 的特性
- 用以支持 Singletons 的 C++ 基本手法
- 如何更好地厉行“Singleton 的唯一性”
- 摧毁 Singleton 并检测摧毁之后的访问动作
- 实现 Singleton 对象的生命期高阶管理方案
- 多线程 (multithreading) 相关问题

我们将逐步找出解决以上问题的相关技术, 最后运用这些技术实现一个泛化的 (通用的) SingletonHolder class template。

Singleton 设计模式并不存在所谓“最佳”实作方案。各种实作技术, 包括不具可移植性的那些方案, 都只不过是适合解决手中的具体问题。本章遵循 *policy-based* (第 1 章) 设计方式, 在一个泛型架构上开发一组实作码 SingletonHolder, 其中甚至为扩展和定制 (customizations) 预留了相应机制。

本章结束前我们会开发出一个 `SingletonHolder` class template，它能生成很多不同类型的 Singletons。`SingletonHolder` 让你精细控制 Singleton 对象如何分配、何时摧毁、在多线程环境下是否安全（所谓 `thread safe`）、被摧毁后如果客户还使用它会发生什么事… `SingletonHolder` class template 可以在任何用户自定义型别上提供“与 Singleton 相关”的服务和机能。

6.1 静态数据 + 静态函数 != Singleton

乍看之下 Singleton 好像没有必要，它似乎很容易以静态成员函数 + 静态成员变量来取代：

```
class Font { ... };
class PrinterPort { ... };
class PrintJob { ... };

class MyOnlyPrinter {
public:
    static void AddPrintJob(PrintJob& newJob)
    {
        if (printQueue_.empty() && printingPort_.available()) {
            printingPort_.send(newJob.Data());
        }
        else {
            printQueue_.push(newJob);
        }
    }
private:
    // All data is static
    static std::queue<PrintJob> printQueue_;
    static PrinterPort printingPort_;
    static Font defaultFont_;
};

PrintJob somePrintJob("MyDocument.txt");
MyOnlyPrinter::AddPrintJob(somePrintJob);
```

但是在某些情况下，这个解法²³有很多缺点。最主要的问题是，静态函数不能成为虚函数，这么一来，如果不开放 `MyOnlyPrinter` 源码，就很难让外界改变其行为。

这种做法的更隐蔽问题是，它使得初始化（initialization）和清理（cleanup）工作变得困难。`MyOnlyPrinter` 的数据在初始化和清理时缺乏中心点。初始化和清理不是一件轻而易举的事，例如我们可以根据 `printingPort_` 的速度来决定 `defaultFont_`。

因此，Singleton 的实作重点主要集中在“产生和管理一个独立对象”上，而且不允许产生另一个这样的对象。

²³ 这段代码实际上演示了另一种模式：Monostate（Ball and Crawford 1998）。

6.2 用以支持 Singletons 的一些 C++ 基本手法

通常，在 C++ 中，Singletons 系通过以下手法的某些变化加以实现：

```
// Header file Singleton.h
class Singleton {
public:
    static Singleton* Instance() { // Unique point of access
        if (!pInstance_)
            pInstance_ = new Singleton;
        return pInstance_;
    }
    ... operations ...
private:
    Singleton(); // Prevent clients from creating a new Singleton
    Singleton(const Singleton&); // Prevent clients from creating
        // a copy of the Singleton
    static Singleton* pInstance_; // The one and only instance
};

// Implementation file Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
```

由于构造函数都是 `private`，客户端无法产生 Singletons，但 Singleton 自身的成员函数——更明确地说是 `Instance()`——则没有这个限制。因此，Singleton 对象的唯一性在编译期就得以实施。这是 C++ 实现 Singleton 设计模式的精髓所在。

如果 Singleton 对象从未被使用（从来没有人调用 `Instance`），也就不会被产生出来。这样的优化成本是，`Instance` 函数起始处需要一个（通常微不足道的）检测动作。如果 Singleton 的产生很昂贵但本身又很少被使用，这种“第一次被需求时才诞生”的方案就愈发显出其优点。

将前例的 `pInstance_` 指针替换为一个完整的 Singleton 对象，藉此简化事情，是一种不幸的诱惑：

```
// Header file Singleton.h
class Singleton {
public:
    static Singleton* Instance() { // Unique point of access
        return &instance_;
    }
    int DoSomething();
private:
    static Singleton instance_;
};

// Implementation file Singleton.cpp
Singleton Singleton::instance_;
```

这不是个好做法。虽然 `Instance_`（角色如同前例的 `pInstance_`）是 `Singleton` 的静态成员，但这两个版本有个重要差异：`Instance_` 被动态初始化（通过执行期间的 `Singleton` 构造函数调用），`pInstance_` 则受益于静态初始化（其型别并无构造函数可通过编译期常量来初始化）。

要知道，程序的第一条 `assembly`（汇编语言）语句被执行之前，编译器就已经完成了静态初始化（通常静态初始化相关数值或动作（`static initializers`）就位于“内含可执行程序”的文件中，所以，程序被装载（`loading`）之际也就是初始化之时）。然而面对不同编译单元（`translation unit`；大致上你可以把编译单元视为可被编译的 C++ 源码文件）中的动态初始化对象，C++ 并未定义其间的初始化顺序，这就是麻烦的主要根源。请看：

```
// SomeFile.cpp
#include "Singleton.h"
int global = Singleton::Instance()->DoSomething();
```

由于无法确保编译器一定先将 `instance_` 初始化，所以全局变量 `global` 的初值设定式中对外 `Singleton::Instance` 的调用有可能传回一个尚未构造的对象。这意味着你无法保证任何外部对象所使用的 `instance_` 是一个已被正确初始化的对象。

6.3 实施“Singleton 的唯一性”

某些语言相关技术对于实施 `Singleton` 的唯一性很有帮助。我们已经用了其中一些：将 `default` 构造函数和 `copy` 构造函数声明为 `private`。后者是为了防止使用者写出这样的代码：

```
Singleton sneaky(*Singleton::Instance()); // error!
// Cannot make 'sneaky' a copy of the (Singleton) object
//      returned by Instance
```

要知道，如果你自己没有定义 `copy` 构造函数，编译器会帮你定义一个 `public` 版本（Meyers 1998a）。如果你明白声明了 `copy` 构造函数，就可以禁止编译器那么做；如果你将构造函数放在 `private` 区段，以上定义 `sneaky` 时就会产生编译期错误（这正是我们要的结果）。

另一个小小改进是：让 `Instance()` 传回 `reference` 而非指针。如果传回指针，调用端（接收端）有可能将它 `delete` 掉。为了将这种可能性降至最低，传回 `reference` 比较更安全些：

```
// inside class Singleton
static Singleton& Instance();
```

编译器暗自生成的另一个成员函数是 `assignment`（赋值）操作符。“唯一性”和“赋值”并没有直接关系，但唯一性会带来一个明显后果：你不能将唯一的 `Singleton` 对象赋予（指派）给另一个对象，因为不可以存在两个 `Singleton` 对象。对 `Singleton` 对象来说，任何赋值动作都是对自身赋值，没有任何意义，所以值得将 `assignment`（赋值）操作符禁掉（做法是：将它声明为 `private` 并且根本不实作它）。

最后一个保护措施是：将析构函数声明为 `private`。有了这个措施，拥有 `Singleton` 对象指针者，就不会（无法）意外删除之。

增加以上所有措施之后，`Singleton` 的接口看起来像这样：

```
class Singleton
{
    static Singleton& Instance();
    ... operations ...
private:
    Singleton();
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
    ~Singleton();
};
```

6.4 摧毁 Singleton

正如前面所说的那样，`Singleton` 是在被需求时才产生的，也就是在 `Instance()` 第一次被调用时产生出来的，因此确立了构造时机，但析构问题没有解决：`Singleton` 应该在什么时候摧毁自身实体？GoF 著作中并没有讨论这个主题，但正如 John Vlissides 在《*Pattern Hatching*》（1998）一书所证实的那样，这个问题很棘手。

事实上就算 `Singleton` 未被删除，也不会造成内存泄漏（memory leak），因为只有当你分配了累积性数据（accumulating data）并丢失对它的所有 `reference` 时，内存泄漏才会发生。这里并不属于上述情况，因为没有什么累积性的东西，而且直到程序结束前我们还保存着对我们所分配的内存的相关认识。此外，当一个进程（process）终止时，所有现代化操作系统都能够将进程所用的内存完全释放。（什么是内存泄漏，什么不是，请见《*Effective C++*》[Meyers 1998a] 条款 10 的相关讨论）

然而泄漏还是存在的，而且更隐蔽更有害，那就是资源泄漏（resource leak）。这是因为 `Singleton` 构造函数可以（可能）索求广泛的资源：网络连接、OS 互斥体（mutexes）和进程通讯（IPC）方法中的各种 `handles`、进程外的 CORBA 或 COM 对象的 `reference`，等等。

避免资源泄漏的唯一正确做法是在程序关闭（结束）时期删除 `Singleton` 对象。问题在于我们必须谨慎选择删除时机，确保 `Singleton` 对象被摧毁后不会再有任何人去取用它。

摧毁 `Singleton` 的最简单方案是仰赖语言机制。以下代码演示 `Singleton` 的另一种实作法。其中 `Instance()` 并未使用动态分配和静态指针，而是使用了一个局部静态变量：

```
Singleton& Singleton::Instance() {
    static Singleton obj;
    return obj;
}
```

这一简单优雅的手法由 Scott Meyers（Meyers 1996a，条款 26）最先提出，所以被称为 *Meyers*

singleton。它依赖某些编译器的神奇技巧。要知道, 函数内的 `static` 对象在该函数第一次执行时被初始化。请不要把“执行期才初始化”的 `static` 变量和“通过编译期常量加以初始化”的基本型 `static` 变量混淆了。例如:

```
int Fun() {
    static int x = 100;    // 译注: 这就是“通过编译期常量加以初始化”
    return ++x;
}
```

这种情况下 `x` 的初始化会在程序中的任何一行被执行之前完成, 而且通常是在程序装载期间。`Fun()` 第一次被调用前, `x` 早就被设为 100 了。但如果初始值不是一个编译期常量, 抑或静态变量是个拥有构造函数的对象, 那么变量的初始化将发生于执行期, 也就是执行流程第一次行经其定义式之时。

此外, 编译器会产生一些代码, 使得初始化之后, 执行期相关机制会登记需被析构的变量。这些代码的 C++ 形式的伪码 (pseudo code) 看起来像下面这样 (两个底线起头的变量应被视为隐藏变量, 也就是由编译器产生和管理的变量):

```
Singleton& Singleton::Instance() {
    // Functions generated by the compiler
    extern void __ConstructSingleton(void* memory);
    extern void __DestroySingleton();
    // Variables generated by the compiler
    static bool __initialized = false;
    // Buffer that holds the singleton
    // (We assume it is properly aligned)
    static char __buffer[sizeof(Singleton)];
    if (!__initialized) {
        // First call, construct object
        // Will invoke Singleton::Singleton
        // In the __buffer memory
        __ConstructSingleton(__buffer);
        // register destruction
        atexit(__DestroySingleton);
        __initialized = true;
    }
    return *reinterpret_cast<Singleton*>(__object);
}
```

其中的核心动作是对 `atexit()` 的调用。`atexit()` 由标准 C 程序库提供, 让你得以注册一些在程序结束之际自动被调用的函数, 且其被调用次序为后进先出 (LIFO)。根据定义, C++ 对象析构以 LIFO 方式进行, 先产生的对象后摧毁。当然, 以 `new` 和 `delete` 管理的对象不遵守这一规则)。`atexit()` 的标记式 (signature) 是:

```
// Takes a pointer to function
// Returns 0 if successful, or
// a nonzero value if an error occurs
int atexit(void (*pFun)());
```


编译器会自动产生出 `__DestroySingleton` 函数（它被执行后会摧毁 `__buffer` 内存内的 Singleton 对象）并将其地址传给 `atexit()`。

`atexit()` 如何运作？每次被调用，它的参数会被压入 C runtime library 所维护的一个私有 stack 内。程序结束之际，执行期相关机制便会调用那些经由 `atexit()` 登记的函数。

很快我们会看到，在 C++ 中，`atexit()` 和 Singleton 设计模式的实现有着重要（有时是不幸）的联系。不论你是否喜欢，它都会伴随我们直至本章结束。无论我们采用哪一种方案来摧毁 Singletons，都必须处理好 `atexit()`，否则后果会让程序员大感意外。

Meyers singleton 提供的是“程序结束之际摧毁 Singleton”的最简单方式。大多数情况下它都能有效运作。下面我将分析它的可能问题，并针对特殊情况提供某些改进，以及其他实作手法。

6.5 Dead（失效的）Reference 问题

为了让讨论更具体，我以一个例子贯穿本章剩余篇幅，用以验证各种实作法。这个例子具有和 Singleton 模式一样的特征：易于表达和理解，但难以实现。

假设有个程序使用了三个 Singletons: `Keyboard`、`Display` 和 `Log`。前二者分别模塑其所对应的真实物体，`Log` 用于错误报告，具体可以是一个文本文件（text file）、或第二主控台（second console），或甚至嵌入式系统中的 LCD 屏幕上的滚动字幕。

假设 `Log` 构造过程需要一定的额外开销，因此最好在错误出现时才将 `Log` 具现化。这么一来，万一程序执行过程中没有任何错误发生，`Log` 就根本不被产生。

程序会向 `Log` 报告 `Keyboard` 或 `Display` 具现化时产生的任何错误。`Log` 也会收集 `Keyboard` 或 `Display` 被摧毁时所（可能）发生的错误。

如果我们以 *Meyers singletons* 实现上述三者，程序并不正确。举个例子，假设 `Keyboard` 成功构造之后 `Display` 初始化失败，于是 `Display` 的构造函数会产生一个 `Log` 记录错误，而且程序准备结束。此时语言规则发挥作用：执行期相关机制会摧毁局部静态对象，摧毁次序和生成次序相反。因而 `Log` 会在 `Keyboard` 之前被摧毁。但万一 `Keyboard` 关闭失败并向 `Log` 报告错误，`Log::Instance()` 会不明就理地回传一个 reference，指向一个已被摧毁的 `Log` 对象的“空壳”。于是程序步入了“行为不确定”的阴暗国土。这就是所谓的 **dead-reference** 问题。

`Keyboard`、`Log` 和 `Display` 的构造和析构次序事先无法预知。我们希望 `Keyboard` 和 `Display` 遵循 C++ 规则（后产生者先摧毁），同时让 `Log` 免于这条规则。我们希望无论 `Log` 何时创建，它一定得在 `Keyboard` 和 `Display` 之后被摧毁，这样才能收集二者析构时的错误报告。

如果程序中使用多个互有关联的 Singletons，我们就无法提供某种自动化方法来控制它们的寿命。一个设计合理的 Singleton 至少应该执行“dead-reference 检测”。为做到这一点，我们可以藉由一个成员变量 `static bool destroyed_` 来追踪析构行为；其值一开始为 `false`，Singleton 析构函数会将它设为 `true`。

实际动手之前让我们再做一次全面检查。除了“产生 Singleton 对象并传回其 reference”之外，Singleton::Instance() 如今身兼另一项任务——检测 dead reference。让我们采用所谓“一个函数一项职责”的设计原则，因此定义出三个独立的成员函数：Create() 高效产生 Singleton 对象；OnDeadReference() 负责处理错误；众所周知的 Instance() 则用以访问唯一的 Singleton 对象。这其中只有 Instance() 被声明为 public。

现在让我们实现这个可执行“dead-reference 检测”的 Singleton。首先在 Singleton class 中增加一个名为 destroyed_ 的 static bool 成员变量，用以检测 dead reference。然后修改 Singleton 析构函数，将 pInstance_ 设为 0，并将 destroyed_ 设为 true。以下是实作码：

```
// Singleton.h
class Singleton
{
public:
    static Singleton& Instance() {
        if (!pInstance_) {
            // Check for dead reference
            if (destroyed_) {
                OnDeadReference();
            }
            else {
                // First call--initialize
                Create();
            }
        }
        return *pInstance_;
    }

private:
    // Create a new Singleton and store a
    // pointer to it in pInstance_
    static void Create(); {
        // Task: initialize pInstance_
        static Singleton theInstance;
        pInstance_ = &theInstance;
    }
    // Gets called if dead reference detected
    static void OnDeadReference() {
        throw std::runtime_error("Dead Reference Detected");
    }
    virtual ~Singleton() {
        pInstance_ = 0;
        destroyed_ = true;
    }
}
```

```
// Data
static Singleton *pInstance_;
static bool destroyed_;
... disabled 'tors/operator= ...
};
```

```
// Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
bool Singleton::destroyed_ = false;
```

很好，可以正常运作！只要程序结束，Singleton 的析构函数就会被调用，于是将 pInstance_ 设为 0 并将 destroyed_ 设为 true。如果此后有某个寿命更长的对象试图取用这个 singleton，执行流程会到达 OnDeadReference()，于是抛出一个型别为 runtime_error 的异常。这个方案廉价、简单，而且不失效率。

6.6 解决 Dead Reference 问题 (I) : Phoenix Singleton

如果将上一节方案应用于 KDL (Keyboard、Display、Log) 问题，结果并不令人满意。Log 被摧毁之后，如果 Display 析构函数需要报错，Log::Instance() 会抛出异常。然而这只是消除 C++ Standard 中的“未确定 (undefined)”行为；现在我们得处理“未令人满意”的行为。

我们希望 Log 无时无刻不存在，不论它当初何时构造。极端情况下我们甚至需要再次产生 Log（尽管它已被摧毁），这样就可以随时用它。这就是 Phoenix Singleton 设计模式背后的思想。

就像传说中的凤凰 (phoenix，又称长生鸟) 可以不断从它自己的焚灰中重生一样，Phoenix Singleton 能够在被摧毁之后复活。我们依然保证任何时候 Singleton 对象只有唯一实体，但如果检测到 dead reference，就可以再次建立该实体。有了 Phoenix Singleton，我们可以轻易解决 KDL 问题：Keyboard 和 Display 还是保持“一般的”Singletons，Log 则成为 Phoenix Singleton。

带有静态变量的 Phoenix Singleton，其实作手法非常简单。一旦检测到 dead reference，我们便会在旧躯壳中产生一个新的 Singleton 对象 (C++ 保证这种可能，因为静态对象的内存在整个程序生命期间都会保留着)。我们也通过 atexit() 登记这个新对象的析构函数。我们不必修改 Instance()，仅有的修改出现在 OnDeadReference() 函数中。

```
class Singleton
{
    ... as before ...
    void KillPhoenixSingleton(); // Added
};

void Singleton::OnDeadReference()
{
    // Obtain the shell of the destroyed singleton
    Create();
}
```

```

// Now pInstance_ points to the "ashes" of the singleton
// - the raw memory that the singleton was seated in.
// Create a new singleton at that address
new(pInstance_) Singleton;
// Queue this new object's destruction
atexit(KillPhoenixSingleton);
// Reset destroyed_ because we're back in business
destroyed_ = false;
}

void Singleton::KillPhoenixSingleton()
{
    // Make all ashes again
    // - call the destructor by hand.
    // It will set pInstance_ to zero and destroyed_ to true
    pInstance_->~Singleton();
}

```

OnDeadReference() 所使用的 new 操作符是所谓 placement new 操作符，它并不分配内存，而是在某个地址上（本例的 pInstance_）构造一个新对象。关于 placement new 操作符的有趣讨论，请参考 Meyers（1998b）。

上面的 Singleton 增加了一个新成员函数 KillPhoenixSingleton()。既然我们通过 new 来令 Phoenix Singleton 重生，我们就不再能够像对待静态变量那样靠编译器技巧摧毁它。我们既然手工建造了它，就必须手工摧毁它，atexit(KillPhoenixSingleton) 保证了这一点。

让我们来分析所有事件的流程。程序结束过程中，Singleton 析构函数被调用，于是将指针重置为 0 并将 destroyed_ 设为 true。现在假设某个全局对象想再次取用 Singleton，于是 Instance() 调用 OnDeadReference()，后者使 Singleton 复活并注册 KillPhoenixSingleton()，然后 Instance() 成功传回一个合法的 reference 指向重生的 Singleton 对象。这一过程有可能重复。

Phoenix Singleton class 确保全局对象及其他 Singletons 能够随时存在一份合法实体。这保证当我们需要一个稳健的全方位对象（如本例的 Log）时，Phoenix Singleton 能够成为一个受人喜爱的方案。如果我们令 Log 为一个 Phoenix Singleton，那么无论错误以何种次序发生，程序都能够正常运作。

atexit 身上的问题

如果将前一节代码和 Loki 实际代码作比较，你会发现一个不同之处：对 atexit 的调用动作被一个 #ifdef 预处理指令包围了起来：

```

#ifdef ATEXIT_FIXED
    // Queue this new object's destructor
    atexit(DeletePhoenixSingleton);
#endif

```

如果你使用 Loki 时没有 `#define ATEXIT_FIXED`，新产生的 Phoenix Singleton 将不会被摧毁，因而造成泄漏——这是我们努力希望避免的。

这个措施和 C++ *Standard* 中一个令人遗憾的疏漏有关。假设你在某个调用中通过 `atexit` 登记某函数，而该调用是由另一个 `atexit` 登记行为造成的，那么 C++ *Standard* 并未说明会发生什么事。为解释这个问题，我写了一个简短的测试程序：

```
#include <cstdlib>
void Bar() {
    ...
}
void Foo() {
    std::atexit(Bar);
}
int main() {
    std::atexit(Foo);
}
```

这个小程序通过 `atexit()` 登记 `Foo`。后者调用 `atexit(Bar)`。这种情况下 C 和 C++ 标准规格书都没有说明会发生什么事。由于 `atexit()` 和静态变量的析构如影随形，所以一旦“程序结束过程”开始进行，我们就会处于一种不稳定状态。C 和 C++ 标准规格都自相矛盾，它们说 `Bar` 会在 `Foo` 之前被调用，因为 `Bar` 较晚才被登记；但是当 `Bar` 被登记时，它已经无法做到“先被调用”，因为此时 `Foo` 已经（正在）被调用。

这个问题听起来会不会太过学究？那么让我以另一种方式来说：直至本书写作之际，在三个应用广泛的编译器上，它带来轻则出错（资源泄漏）重则造成程序崩溃²⁴的后果。

面对这个问题，编译器得花点时间找出补救方案。直至目前 Loki 中的那个宏（指前述那个 `#ifdef`）还得摆在那儿。在某些编译器上，如果你第二次产生一个 Phoenix Singleton，它最终会造成内存泄漏。视你手上编译器说明文档中的描述而定，或许你会想要在含入 `Singleton.h` 头文件之前，先定义（`#define`）`ATEXIT_FIXED`。

6.7 解决 Dead Reference 问题 (II)：带寿命的 Singletons

Phoenix Singletons 能够满足很多场合的需求，但它也有缺点：它破坏了 Singleton 正常的生命周期，从而可能给使用者带来迷惑。如果你的 Singleton 持有状态（state），那个状态会在“析构 - 创建”周期中丢失。如果要经由 Phoenix 策略来实现 concrete Singleton，实作者就必须特别注意

²⁴ 经由新闻群组 `comp.std.c++` 和电子邮件，我曾经就 `atexit` 的现状和 ANSI/ISO C++ 标准委员会主席 Steve Clamage 进行讨论。他对这个问题十分清楚，并已针对 C9X 和 C++ 提供了一份错误修改报告。这份报告可在以下网址找到：<http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/lwg-issues.html#3>。幸运的是目前提出的解决方案采用的正是本章所讨论的 Singleton 实作法：即使对于结束期间被呼叫的函数，`atexit` 还是以 stack 方式运作，这正是我们所希望的。本书写作之时，这一方案已被批准并准备写入标准规格中。

保持“析构”和“重新构造”两个时刻间的状态。

这很令人讨厌，特别是在某些情况下（例如前述的 KDL 例子）你其实知道次序应该怎样：无论如何如果创建了 `Log`，它就必须在 `Keyboard` 和 `Display` 之后摧毁。换句话说，在这种情况下 `Log` 必须拥有“比 `Keyboard` 和 `Display` 更长”的寿命。我们希望有一种简单方法来控制各种 `Singletons` 的寿命。果真如此就可以赋予 `Log` “比 `Keyboard` 和 `Display` 更长”的生命，从而解决 KDL 问题。

等一下，我还没说完：这个问题不仅适用于 `Singletons`，也适用于全局对象。这里说的是“寿命控制（longevity control）”概念与 `Singleton` 无关：对象的寿命愈长，就愈晚被摧毁。至于主角是 `Singleton` 对象或某个动态分配的全局对象，无关紧要。我们需要写出这样的代码：

```
// This is a Singleton class
class SomeSingleton { ... };
// This is a regular class
class SomeClass { ... };

SomeClass* pGlobalObject(new SomeClass);

int main()
{
    SetLongevity(&SomeSingleton().Instance(), 5);
    // Ensure pGlobalObject will be deleted
    // after SomeSingleton's instance
    SetLongevity(pGlobalObject, 6);
    ...
}
```

`SetLongevity()` 接受两个参数，一个是 **reference**，指向任意型别对象，另一个是整数值，代表寿命：

```
// Takes a reference to an object allocated with new and
// the longevity of that object
template <typename T>
void SetLongevity(T* pDynObject, unsigned int longevity);
```

这个函数保证，和其他所有寿命较短的对象相比，`pDynObject` 存在的时间比较长。当程序结束时，所有通过 `SetLongevity()` 登记的对象便会根据寿命长短被依序删除。

对那些“寿命受编译器控制”的对象（例如一般全局对象、`static` 对象、`auto` 对象）来说，你无法运用 `SetLongevity()`。编译器已经自动产生了一些代码来摧毁这些对象，如果你又调用 `SetLongevity()`，它们就会被摧毁两次（这对程序绝无好处）。`SetLongevity` 针对的只是“经由 `new` 分配而得”的对象。此外，对某个对象调用 `SetLongevity()`，表示你不会对那个对象调用 `delete`（你必须遵守这条约定）。

另一种选择是建立一个“依存性管理器 (dependency manager)”，一个控制“对象之间彼此依存性”的对象。这东西会开放一个泛型函数 `SetDependency()`，如下所示：

```
class DependencyManager
{
public:
    template <typename T, typename U>
    void SetDependency(T& dependent, U& target);
    ...
};
```

其析构函数会以一定的次序摧毁对象，并在摧毁目标物之前先摧毁它们的依存性。

以 `DependencyManager` 为本的方案有一个很大的缺点：依存性两端的对象都必须存在。这意味着如果你想在 `Keyboard` 和 `Log` 之间建立依存性，你就必须拥有 `Log` 对象——即使将来根本不需要它。

为避免这个问题，我们可以在 `Log` 构造函数中建立 `Keyboard` 和 `Log` 的依存性，但是这会将 `Keyboard` 和 `Log` 之间的耦合关系加剧到令人无法接受的程度：`Keyboard` 取决于 `Log` 的定义（因为 `Keyboard` 要使用 `Log`），`Log` 也取决于 `Keyboard` 的定义（因为 `Log` 要设置与 `Keyboard` 之间的依存性）。这是一种循环依存 (circular dependency)。本书第 10 章有一些详尽的阐述，告诉你应当避免循环依存。

让我们回到对象寿命的思考范围。由于 `SetLongevity()` 必须和 `atexit()` 相处融洽，所以我们必须认真定义这两个函数间的关系。例如下面这个程序中，让我们确定析构函数调用的确切顺序：

```
class SomeClass { ... };

int main()
{
    // Create an object and assign a longevity to it
    SomeClass* pObj1 = new SomeClass;
    SetLongevity(*pObj1, 5);
    // Create a static object whose lifetime
    // follows C++ rules
    static SomeClass obj2;
    // Create another object and assign a greater
    // longevity to it
    SomeClass* pObj3 = new SomeClass;
    SetLongevity(*pObj3, 6);
    // How will these objects be destroyed?
}
```

`main` 之中既定义了“带寿命的对象”，也定义了“遵循 C++ 规则的对象”。为这三个对象定义一个合理的析构顺序很困难，因为除了使用 `atexit()`，我们没有任何方法可以操控那个由执行期机制维护的隐藏性 `stack`。

仔细分析了各个约束条件（constraints）后，我们得出以下设计决定：

- 每一个 `SetLongevity()` 调用动作都产生一个 `atexit()` 调用动作。
- 短寿对象的析构行为发生在长寿对象的析构行为之前。
- 寿命相同的对象，其析构遵循 C++ 规则：后构造者先被摧毁。

这些规则会在先前的范例程序中保证这样的析构次序：`*pObj1`, `obj2`, `*pObj3`。第一次调用 `SetLongevity()` 会产生一个 `atexit` 调用动作，用于 `*pObj3` 的摧毁，第二个调用动作会相应发出一个 `atexit` 调用动作，用于 `*pObj1` 的摧毁。

`SetLongevity()` 赋予开发者“管理对象寿命”的强大能力；面对“与对象寿命相关”的 C++ 内建规则，它也提供了良好而又合理的互动方式。但是请注意，和其他许多功能强大的工具一样，它也会带来危险。使用它的经验法则是：任何对象 A 如果使用了带寿命的对象 B，A 的寿命就必须短于 B 的寿命。

6.8 实现“带寿命的 Singletons”

一旦为 `SetLongevity()` 定义了完整规格，实现起来就不再那么复杂。`SetLongevity()` 维护一个隐藏的 `priority queue`，这个 `queue` 和无法被应用程序访问的那个 `atexit stack` 是分开的。`SetLongevity()` 会调用 `atexit()`，并总是传给它同一个函数指针——该函数从 `stack` 中取出一个元素并删除之。

这里的要素在于 `priority queue`。传给 `SetLongevity()` 的寿命值用于建立优先权（`priority`）；对于某个已知寿命值而言，这个 `queue` 的行为像个 `stack`。相同寿命的对象，其析构顺序遵循“后进先出”原则。尽管从名称看来 `std::priority_queue` 似乎合用，但这儿不能用它，因为面对相同优先权的元素，它无法保证正确处理次序。

这个数据结构保存的元素都是指针，指向 `LifetimeTracker` 型别，其接口由一个虚析构函数和一个 `comparison`（比较）操作符组成；其派生类必须改写该析构函数。此外，还有一个 `friend Compare()`，稍后你会看到它的用途。

```
namespace Private {
    class LifetimeTracker {
    public:
        LifetimeTracker(unsigned int x) : longevity_(x) {}
        virtual ~LifetimeTracker() = 0;
        friend inline bool Compare(
            unsigned int longevity,
            const LifetimeTracker* p)
        { return p->longevity_ < longevity; }
    private:
        unsigned int longevity_;
    };
}
```



```
// Definition required
inline LifetimeTracker::~LifetimeTracker() {}
}
```

`priority queue` 是一个简单的动态 array，所有元素都是指针，指向 `LifetimeTracker`：

```
namespace Private {
    typedef LifetimeTracker** TrackerArray;
    extern TrackerArray pTrackerArray;
    extern unsigned int elements;
}
```

`Tracker` 型别只有一个实体。因此，`pTrackerArray` 会面临刚才讨论的所有 Singleton 问题。我们会陷入“鸡生蛋、蛋生鸡”的有趣问题中：`SetLongevity()` 必须随时可用，还得管理私有空间。为了处理这个问题，`SetLongevity()` 借助 `std::malloc` 家族 (`malloc`、`realloc`、`free`) 中的低阶函数来仔细操控 `pTrackerArray`²⁵。这么一来，我们就将“鸡生蛋、蛋生鸡”的问题转给了 C heap 分配器——幸运的是在程序整个生命期内，这个分配器都可以正常工作。至此，`SetLongevity` 的实现很简单：产生一个 concrete tracker 对象，将它加入 `stack`，并通过 `atexit` 登记某个函数。

以下代码朝向“泛化”迈出了重要的一步，它引入一个仿函数 (functor) 用以管理“正被摧毁”的被追踪物。其基本原理是，你并非总是以 `delete` 释放对象；它可能被分配于另一个 heap 中。缺省情况下“摧毁器 (destroyer)”是一个函数指针，该函数会调用 `delete`。缺省函数名为 `Delete()`，根据“被删除物”之型别而被模板化 (templated)。

```
template <typename T> void Delete(T* pObj)
{
    delete pObj;
}

// 译注：此与 Loki 源码略有不同，请注意

namespace
{
    // Concrete lifetime tracker for objects of type T
    template <typename T, typename Destroyer>
    class ConcreteLifetimeTracker : public LifetimeTracker
    {
    public:
        ConcreteLifetimeTracker(T* pDynObject,
                                unsigned int longevity, Destroyer destroyer)
            : LifetimeTracker(longevity), pTracked_(pDynObject),
              destroyer_(destroyer)
        {}
    };
}
```

²⁵ 事实上，`SetLongevity` 只使用 `std::realloc`。`realloc()` 可取代 `alloc()` 和 `free()`：如果你通过一个 `null` 指针来调用它，它动起来就像 `std::alloc`；如果你通过大小为零的参数来调用它，它动起来就像 `std::free`。基本上 `std::realloc` 是“以 `malloc` 为主”的一种全能分配函数。

```

    ~ConcreteLifetimeTracker()
    {
        destroyer_(pTracked_);
    }
private:
    T* pTracked_;
    Destroyer destroyer_;
};

void AtExitFn(); // Declaration needed below
}

template <typename T, typename Destroyer>
void SetLongevity(T* pDynObject, unsigned int longevity,
    Destroyer d = Private::Deleter<T>::Delete)
{
    TrackerArray pNewArray = static_cast<TrackerArray>(
        std::realloc(pTrackerArray, sizeof(T) * (elements + 1)));
    if (!pNewArray) throw std::bad_alloc();
    pTrackerArray = pNewArray;
    LifetimeTracker* p = new ConcreteLifetimeTracker<T, Destroyer>(
        pDynObject, longevity, d);
    TrackerArray pos = std::upper_bound(
        pTrackerArray, pTrackerArray + elements, longevity, Compare);
    std::copy_backward(pos, pTrackerArray + elements,
        pTrackerArray + elements + 1);
    *pos = p;
    ++elements;
    std::atexit(AtExitFn);
}

```

像 `std::upper_bound` 和 `std::copy_backward` 这类东西, 你得花点时间才能习惯, 但它们的确可以让原本不简单的代码比较易于编写和阅读。上述函数将一个新产生的、指向 `ConcreteLifetimeTracker` 的指针插入 `pTrackerArray` 所指的 `array` 中, 并使之保持正确次序, 此外我还处理了错误和异常。

现在 `LifetimeTracker::Compare()` 的用途很清楚了。 `pTrackerQueue` 所指的 `array` 以寿命长短来排序。长寿对象靠近 `array` 头部, 相同寿命的对象则依其插入顺序排列。 `SetLongevity()` 可以确保这一切。

`AtExitFn()` 传回最短寿命对象 (位于 `array` 尾端) 并删除之。删除一个“指向 `LifetimeTracker`”的指针, 会调用 `ConcreteLifetimeTracker` 析构函数, 其内删除被追踪的对象:

```

static void AtExitFn()
{
    assert(elements > 0 && pTrackerArray != 0);
    // Pick the element at the top of the stack
    LifetimeTracker* pTop = pTrackerArray[elements - 1];
    // Remove that object off the stack
    // Don't check errors~realloc with less memory
}

```

```

// can't fail
pTrackerArray = static_cast<TrackerArray>(std::reallocate(
    pTrackerArray, sizeof(T) * --elements));
// Destroy the element
delete pTop;
}

```

撰写 `AtExitFn()` 时需得小心一些：它必须取出（`pop`）`stack` 顶部元素并删除之。在那个元素的析构函数中，受管对象会被删除。这里的一个小窍门是 `AtExitFn()` 必须先将 `stack` 顶端对象取出，然后再删除之，因为摧毁某个对象时可能会产生另一个对象，从而造成另一个元素被压入 `stack`。虽然这看起来不大可能，但是当 `keyboard` 析构函数要使用 `Log` 时，这却是事实。

带寿命的 `Singletons` 可以通过以下方式使用 `SetLongevity()`：

```

class Log
{
public:
    static void Create()
    {
        // Create the instance
        pInstance_ = new Log;
        // This line added
        SetLongevity(*this, longevity_);
    }
    // Rest of implementation omitted
    // Log::Instance remains as defined earlier
private:
    // Define a fixed value for the longevity
    static const unsigned int longevity_ = 2;
    static Log* pInstance_;
};

```

如果以类似手法实作 `keyboard` 和 `Display`，并将 `longevity_` 定义为 1，那么当 `keyboard` 和 `Display` 被摧毁时，`Log` 一定存在。这就解决了 KDL 问题——哦，是吗？如果你的程序使用多线程（`multiple threads`），那又会怎样？

6.9 生活在多线程世界

`Singleton` 也得处理线程（`threads`）。假设我们有一个刚刚启动的程序，其中有两个线程要访问下面这个 `Singleton`：

```

Singleton& Singleton::Instance()
{
    if (!pInstance_) // 1
    {
        pInstance_ = new Singleton; // 2
    }
    return *pInstance_; // 3
}

```

第一个线程进入 `Instance` 并检测 `if` 条件。由于这是第一次访问，所以 `pInstance_` 为 `null`，于是进入 `//2` 那一行，准备调用 `new` 操作符。此时有可能 OS 调度器（scheduler）中断了这个线程，将控制权转给另一个线程。

第二个线程粉墨登场，调用 `Singleton::Instance()`，并发现 `pInstance_` 为 `null`——因为第一个线程没机会修改它。到目前为止第一个线程只是完成了对 `pInstance_` 的测试。现在假设第二个线程完成了对 `new` 的调用，顺利完成 `pInstance_` 的赋值动作并带走它。

不幸的是，当第一个线程再次苏醒时，它记得它应该执行 `//2` 代码，并因此对 `pInstance_` 再次赋值，并带走它。尘埃落定之后，程序中有了两个 `Singleton` 对象而不是一个，其中一个必定造成内存泄漏。每个线程各自拥有一个独立的 `Singleton` 实体，这肯定会让程序步入混乱之中。这还仅仅是可能情况之一，如果多个线程蜂拥而上地访问其 `Singleton`（想象你正对这个程序调试），会有什么后果？

拥有多线程编程经验的程序员应该会意识到，这是一个典型的竞态条件（race condition）问题。`Singleton` 遇上多线程，是可以预料的，然而 `Singleton` 对象是共享的全局资源，而所有共享的全局资源对竞态条件（race condition）和多线程环境而言都是不可靠的。

“双检测锁定”（Double-Checked Locking）模式

关于 `multithreaded singletons`（多线程单件）的全面论述最早由 Douglas Schmidt（1996）提出。同一篇文章中作者还介绍了一个漂亮的解法：所谓“双检测锁定”（Double-Checked Locking）模式。此方案由 Doug Schmidt 和 Tim Harrison 共同发明。

下面这个平淡无奇的明显解法具有疗效，但不吸引人：

```
Singleton& Singleton::Instance()
{
    // mutex_ is a mutex object
    // Lock manages the mutex
    Lock guard(mutex_);
    if (!pInstance_) {
        pInstance_ = new Singleton;
    }
    return *pInstance_;
}
```

`Lock` class 是一个典型的 `mutex`（互斥体）管理者（关于 `mutexes` 的详细介绍请参阅本书附录）。`Lock` 构造函数为 `mutex` 加锁，析构造函数则为之解锁。当 `mutex_` 被锁定时，其他试图锁定同一个 `mutex` 的线程都必须等待。

这就消除了竞态条件：当某个线程对 `pInstance_` 赋值时，其他所有线程会止步于 `guard` 构造函数中。当另一个线程为 `mutex_` 加锁时，它会发现 `pInstance_` 已被初始化。很好。

然而正确的解法不一定是吸引人的解法。本法之不足在于缺乏效率。每次 `Instance()` 执行都会引发同步对象（`synchronization object`）的加锁与解锁，即使竞态条件只在生命期内出现一次。这些操作通常十分昂贵，比简单的 `if (!pInstance_)` 测试昂贵得多。现今系统中，“测试 - 分向前进”和“关键区段（`critical section`）锁定”的耗时相差高达数个量级。

以下代码也希望能够成为一个解法。它试图避免额外开销：

```
Singleton& Singleton::Instance()
{
    if (!pInstance_) {
        Lock guard(mutex_);
        pInstance_ = new Singleton;
    }
    return *pInstance_;
}
```

现在，额外开销的确不见了，但竞态条件又回来了。第一个线程通过了 `if` 测试，但正当它准备进入“同步区段”时，OS 调度器中断了这个线程并将控制权转给另一个线程。后者通过了 `if` 测试（一点也不令人惊讶，它发现了一个 `null` 指针）并进入同步区段，而后结束执行。当第一个线程重新醒来时，它也进入同步区段，但为时已晚，程序中于是构造出两个 `Singleton` 对象。

这像是个没有答案的脑筋急转弯问题，但事实上的确有一个十分简单优雅的解法——所谓“双检测锁定”。

想法很简单：首先进行检测，然后进入同步码，然后再次检测。但这一次指针要么已被初始化，要么就是 `null`。下面的代码可以帮助你理解和领略“双检测锁定”模式。啊，的确，计算机工程中也存在着“美”。

```
Singleton& Singleton::Instance()
{
    if (!pInstance_)           // 1
    {                           // 2
        Guard myGuard(lock_);  // 3
        if (!pInstance_)       // 4
        {
            pInstance_ = new Singleton;
        }
    }
    return *pInstance_;
}
```

假设某个线程的控制流程进入了模糊区（注释第 2 行），此处可能有数个线程同时进入。但同步区则是“同一时刻只会有一个线程进入”。到了注释第 3 行，模糊不复存在。一切都无比清晰：指针要么已经完全初始化，要么根本没有被初始化。第一个进入的线程会初始化指针变量，其他所有线程都会在注释第 4 行的检测行动中失败，因而不会产生任何东西。

第一个检测快速而粗糙。如果 `Singleton` 对象存在，你可以得到它。否则就需要做进一步检测。第二个检测缓慢而精确：它判断 `Singleton` 是否确实被初始化，如果没有，这个线程就会负责对它初始化。这就是“双检测锁定”模式。现在我们可谓鱼和熊掌兼得：大多数情况下对 `Singleton` 的访问都具备应有的速度，而构造期间也不存在竞态条件。酷毙了。但是…

具有丰富多线程经验的程序员都知道，即使是“双检测锁定”，虽然纸上谈兵时它是正确的，可真正实践时它并不总是正确。RISC（精简指令集）机器的编译器有一个所谓的 `code arranger`，它会重新排列编译器所产生出来的汇编语言指令，使代码能够最佳运用 RISC 处理器的并行（`parallel`）特性（例如 RISC 机器可以同时执行一个 `load` 动作和一个 `add` 动作）。

“重新排列指令”是 RISC 处理器能够达到优化的一个主要因素，它甚至可以使速度加倍。但它也可能破坏“双检测锁定”模式：编译器有可能在锁定 `mutex` 之前先执行第二个 `if(!pInstance_)` 测试，于是竞态条件（`race condition`）再度出现。如果你希望产生出来的代码总是“政策上正确”，那会剧烈降低所有代码的速度。

结论是，实作“双检测锁定”模式之前，你应该先查阅手上的编译器说明文档（这使它成了“三次检测”^②）。通常系统平台会提供选择性的、不可移植的并发基本元素（`concurrency primitives`），例如 `memory barrier`（内存屏障），那是一种轻型的 `mutex`。至少你应该在 `pInstance_` 前添加 `volatile` 饰词，因为合理的编译器会为 `volatile` 对象产生出恰当而明确的代码。

6.10 将一切组装起来

本章论遍 `Singleton` 的各种可能实作法，评价它们的相对优势和弱点。我们的讨论并没有产生唯一一份实作品，因为只有你手中的具体问题才能决定什么是“`Singleton` 最佳实现”。

Loki 定义的 `SingletonHolder` class template 是一种 `Singleton` 容器（`container`），用以协助你运用 `Singleton` 设计模式。`SingletonHolder` 采用 `policy-based` 设计（第1章），可协助制作出“使用者自定义之 `Singleton` 对象”。运用 `SingletonHolder` 时你可以挑选你需要的特性，并可添加你自己的代码。极端情况下你甚至可以一切重头开始——没问题，只要你的确处于极端情况。

本章也讨论了数个相互之间几乎没有关系的主题。那么，`Singleton` 如何才能实现这么多需求，同时又避免程序膨胀呢？答案是：将 `Singleton` 细心分解为数个 `policies`，如同第1章所述，然后我们就可以在短短数行代码中实现先前讨论过的所有情况。经由 `template` 具现化，你可以选择你需要的特性而不必在意你不需要的东西。是的，这一点很重要：这里组装出来的 `Singleton` 实作品并非集大成的 `class`。只有被你挑选的特性才会最终被包含到生成的代码中。此外，这份实作品预留了调整和扩展的余地。

6.10.1 将 SingletonHolder 分解为策略

让我们先界定清楚，前面讨论过的各种实作可以划分出哪些策略。我们可以归纳出创建（生成）、寿命和线程三大问题。这是 Singleton 的三个最重要的开发方向。相应的三个策略如下：

1. **Creation**（创建、生成）。你可以经由各种方式创建 Singleton。通常 **Creation** 策略通过 `new` 操作符来创建（产生）对象。将创建作为独立策略分离开来是必需的，因为这样一来你就可以创建多态对象（polymorphic objects）。
2. **Lifetime**（寿命）。我们可以划分出以下寿命策略：
 - a. 遵循 C++ 规则，后创建者先被摧毁。
 - b. 复现（Phoenix Singleton，不死单件）
 - c. 用户控制（“带寿命的” Singleton）
 - d. 无限生命期（“会造成泄漏的” Singleton，那是一个永远不会被摧毁的对象）
3. **ThreadingModel**（多线程模型）。Singleton 是否为单线程、标准多线程（使用 `mutex` 和“双检验锁定”模式），或是使用了“不具可移植性”的线程模型。

所有 Singleton 实作品都必须保证一个前提，那就是“唯一性”。这不算是策略，因为改变这个前提就会违反 Singleton 的定义。

6.10.2 定义 SingletonHolder 的策略需求

让我们定义出 SingletonHolder 对其策略有哪些必要需求。

Creation 策略必须能够创建和摧毁对象，所以它必须开放两个相应的函数。假设 `Creator<T>` 是一个符合 **Creation** 策略的 class，那么 `Creator<T>` 必须支持以下两个调用动作：

```
T* pObj = Creator<T>::Create();
Creator<T>::Destroy(pObj);
```

请注意，`Create` 和 `Destroy` 必须是 `Creator` 的两个 `static` 成员函数。Singleton 并不拥有 `Creator` 对象，因为那将造成 Singleton 生命期的问题。

从根本来说，**Lifetime** 策略必须对“经由 **Creation** 策略所创建的 Singleton 对象”的析构进行调度。本质上 **Lifetime** 策略的功能应当归结为它“在程序生命期的某一时刻摧毁 Singleton 对象”的能力。此外，如果程序违反 Singleton 对象的寿命规则，**Lifetime** 将决定采取哪些措施。因此：

- 如果有必要依据 C++ 规则来摧毁 Singleton，**Lifetime** 将使用 `atexit()` 类似的机制。
- 面对 Phoenix singleton，**Lifetime** 还是使用类似 `atexit()` 的机制，但允许 Singleton 对象重新生成。
- 对于带寿命的 Singleton，**Lifetime** 会向 `SetLongevity` 发出一个调用，如 6.7 节和 6.8 节所述。
- 对于无限期寿命，**Lifetime** 不会采取任何措施。

归纳而言，**Lifetime** 策略必须提供两个函数：`ScheduleDestruction()` 负责设定合适的析构时间，`OnDeadReference()` 负责在发现 `dead-reference` 时执行某些动作。

如果 `Lifetime<T>` 是一个实现 **Lifetime** 策略的 `class`，那么下面的式子是合理的：

```
void (*pDestructionFunction)();
...
Lifetime<T>::ScheduleDestruction(pDestructionFunction);
Lifetime<T>::OnDeadReference();
```

成员函数 `ScheduleDestruction()` 接受一个函数指针，指向析构操作的实际执行函数。这么一来我们就可以将 **Lifetime** 策略和 **Creation** 策略结合在一起。别忘了，**Lifetime** 不会干预析构方式，那是 **Creation** 的权力；**Lifetime** 的唯一职责是时间调度，也就是决定析构发生时间。

任何情况下 `OnDeadReference()` 都会抛出异常，除非你用的是 `Phoenix singleton`——那么它就什么也不做。

ThreadingModel 策略也就是本书附录所说的那个策略。`SingletonHolder` 不支持 `object-level` 锁定，只支持 `class-level` 锁定。这是因为你永远只有一个 `Singleton` 对象。

6.10.3 组装 SingletonHolder

现在让我们开始定义 `SingletonHolder class template`。正如第 1 章所说，每一种策略都要求拥有一个 `template` 参数。除此之外我们还预留了一个 `template` 参数 `T`——我们要为该型别提供 `singleton` 特性。`SingletonHolder` 本身并不是 `Singleton`，而是为现有的 `classes` 提供 `singleton` 的行为及管理。

```
template
<
    class T,
    template <class> class CreationPolicy = CreateUsingNew,
    template <class> class LifetimePolicy = DefaultLifetime,
    template <class> class ThreadingModel = SingleThreaded
>
class SingletonHolder
{
public:
    static T& Instance();
private:
    // Helpers
    static void DestroySingleton();
    // Protection
    SingletonHolder();
    ...
    // Data
    typedef ThreadingModel<T>::VolatileType InstanceType;
    static InstanceType* pInstance_;
    static bool destroyed_;
};
```


个体变量（instance variable）的型别并非是你所想象的 T ，而是 `ThreadingModel<T>::volatileType`。这一型别定义会被开展为 T 或 `volatile T`，取决于实际的线程模型（threading model）。将 `volatile` 饰词运用于某个型别身上，相当于告诉编译器：那个型别的值有可能被多个线程修改。知道了这一点，编译器就会避免某些优化措施（例如“将数值保存于内部暂存器中”等等），以免导致多线程程序工作紊乱。最安全的选择便是：将 `pInstance_` 定义为 `volatile T*`，既可用于多线程环境，也不会伤及单线程程序。

但是另一方面，在单线程模型中你可能的确想获得那些优化结果，所以 $T*$ 将是 `pInstance_` 的最适当型别。这就是“`pInstance_` 的实际型别要由 `ThreadingModel` 策略来决定”的原因。如果 `ThreadingModel` 为单线程策略，它只不过是像下面这样定义 `volatileType`：

```
template <class T> class SingleThreaded
{
    ...
public:
    typedef T VolatileType;
};
```

多线程策略的定义则会以 `volatile` 来修饰 T 。关于线程模型的更详细介绍，请阅读本书附录。

现在，让我们定义成员函数 `Instance()`，它将三个策略贯穿起来：

```
template <...>
T& SingletonHolder<...>::Instance()
{
    if (!pInstance_)
    {
        typename ThreadingModel<T>::Lock guard;
        if (!pInstance_)
        {
            if (destroyed_)
            {
                LifetimePolicy<T>::OnDeadReference();
                destroyed_ = false;
            }
            pInstance_ = CreationPolicy<T>::Create();
            LifetimePolicy<T>::ScheduleCall(&DestroySingleton);
        }
    }
    return *pInstance_;
}
```

`Instance()` 是 `SingletonHolder` 开放的唯一一个 `public` 函数。它在 `CreationPolicy`、`LifetimePolicy` 和 `ThreadingModel` 之上打造了一层外壳(shell)。`ThreadingModel<T> policy class` 开放了一个内部类：`Lock`。在 `Lock` 对象生命期内，其他所有线程如果想产生任何 `Lock` 对象，都会形成阻塞（*block*，详见附录）。

`DestroySingleton()` 只不过是用来摧毁 `Singleton` 对象, 消除内存, 并将 `destroyed_` 设为 `true`。`SingletonHolder` 绝不会调用 `DestroySingleton()`, 它只是将其地址传给 `LifetimePolicy<T>::ScheduleDestruction`。

```
template <...>
void SingletonHolder<...>::DestroySingleton()
{
    assert(!destroyed_);
    CreationPolicy<T>::Destroy(pInstance_);
    pInstance_ = 0;
    destroyed_ = true;
}
```

`SingletonHolder` 将 `pInstance_` 和 “`DestroySingleton()` 的地址” 传给 `LifetimePolicy<T>`, 意图为 `LifetimePolicy` 提供足够信息, 以实现我们所知道的那些行为: C++ 规则、复现 (Phoenix singleton)、用户控制 (带寿命之 `Singleton`) 和无限生命期。方法如下:

- 遵循 C++ 规则。 `LifetimePolicy<T>::ScheduleDestruction()` 调用 `atexit()`, 将 `DestroySingleton()` 的地址传递过去。 `OnDeadReference()` 会抛出 `std::logic_error` 异常。
- 复现。同上, 但 `OnDeadReference()` 不会抛出异常, `SingletonHolder` 的执行流程会继续, 并重新产生对象。
- 使用者自行控制。 `LifetimePolicy<T>::ScheduleDestruction()` 调用 `SetLongevity(GetLongevity(pInstance))`。
- 无限生命期。 `LifetimePolicy<T>::ScheduleDestruction()` 的实作码为空。

`SingletonHolder` 会处理 `dead-reference` 问题, 以履行 `LifetimePolicy` 的职责。这非常简单: 如果 `SingletonHolder::Instance()` 检测到了一个 `dead reference`, 便调用 `LifetimePolicy::OnDeadReference()`。如果 `OnDeadReference()` 返回, `Instance()` 会重新产生一个新实体。总而言之 `OnDeadReference()` 要么抛出一个异常, 要么如果你不希望具备 Phoenix Singleton 行为, 它就终止程序。对于 Phoenix Singleton, `OnDeadReference()` 什么也不做。

好了, 这就是 `SingletonHolder` 的全部实现。当然, 此刻大量工作委托给了前述三个策略。

6.10.4 “常备策略”之实作

分解为策略很难, 但分解完毕后, 策略很容易实现。让我们归纳一下有哪些 `policy classes` 用来实现常见的 `singleton` 类型。表 6.1 展示了 `SingletonHolder` 预先定义的 `policy classes`, 粗体字所表示的 `policy class` 是为缺省之 `template` 参数。

表 6.1 针对 SingletonHolder 预定义的策略

策略 (policy)	预定义的 class template	说明
Creation	CreateUsingNew	通过 new 操作符和 default 构造函数产生对象
	CreateUsingMalloc	通过 std::malloc 及其 default 构造函数产生对象
	CreateStatic	在静态内存中产生对象
Lifetime	DefaultLifetime	遵循 C++ 规则来管理对象寿命。通过 atexit() 完成工作
	PhoenixSingleton	和 DefaultLifetime 相同，但允许重新产生 Singleton 对象
	SingletonwithLongevity	可对 Singleton 对象赋予寿命。假设存在一个 namespace-level GetLongevity() 函数，接受引数 pInstance_，传回 Singleton 对象寿命
	NoDestroy	不摧毁 Singleton 对象
ThreadingModel	SingleThreaded	请阅读本书附录，获得线程模型的细节说明
	ClassLevelLockable	

接下来我们所需了解的，不过就是如何使用和扩展这个小巧但功能强大的 SingletonHolder template。

6.11 使用 SingletonHolder

SingletonHolder class template 并没有提供“和应用程序相关连”的功能。它只是为另一个 class（亦即本章范例代码中的 T）提供“和 Singleton 相关的”服务。我们称 T 为 client class（客户端，接受服务的类）。

Client class 必须采取一切措施来预防“无人管理”的构造和析构：举凡 default 构造函数、copy 构造函数、assignment 操作符、析构函数、address-of (取址) 操作符都必须声明为 private。

采取了这些保护措施之后,对于你所使用的 Creation policy class,你还得批准其友谊(friendship)关系。如果你有个 class 想要使用 SingletonHolder, 上述保护措施和 friend 声明,是你需要做的一切修改。请注意, 修改项目可以选择, 你的决定将在“触及既有代码所带来的不便性”和“伪实体 (spurious instances) 的风险”之间形成一个折衷。

在一个特定的 Singleton 实作方案中, 设计决策通常反映在以下这样的型别定义 (type definition) 中。就像你以前调用函数时可能会传递标志和选项一样, 这里你也要将标志传给型别定义, 用以选择你想要的行为。

```
class A { ... };
typedef Singleton<A, CreateUsingNew> SingleA;
// from here on you use SingleA::Instance()
```

提供一个 Singleton 并让它“传回 derived class 对象”, 是很简单的事, 我们只需修改 Creator policy class:

```
class A { ... };
class Derived : public A { ... };

template <class T> struct MyCreator : public CreateUsingNew<T>
{
    static T* Create()
    {
        return new Derived;
    }
};

typedef SingletonHolder<A, StaticAllocator, MyCreator> SingleA;
```

类似情况, 你可以为构造函数提供参数, 或使用不同的分配策略。你可以根据每一个策略来调节 Singleton。这样你便可以很大程度地定制 Singleton, 并在必要时刻获得缺省行为的好处。

SingletonWithLongevity policy class 要求你必须定义一个 namespace-level GetLongevity() 函数, 其定义式如下:

```
inline unsigned int GetLongevity(A*) { return 5; }
```

只有当你在 SingleA 的型别定义中用上了 SingletonWithLongevity 时, 这才需要。

复杂的 KDL 问题促使我们提出了很多试探性的实作手法。下面演示的就是如何利用 Singleton class template 来解决 KDL 问题。当然, 这些定义都应该位于它们相应的头文件中。

```
class KeyboardImpl { ... };
class DisplayImpl { ... };
```

```

class LogImpl { ... };

...

inline unsigned int GetLongevity(KeyboardImpl*) { return 1; }
inline unsigned int GetLongevity(DisplayImpl*) { return 1; }
// The log has greater longevity
inline unsigned int GetLongevity(LogImpl*) { return 2; }

typedef SingletonHolder<KeyboardImpl, SingletonWithLongevity> Keyboard;
typedef SingletonHolder<DisplayImpl, SingletonWithLongevity> Display;
typedef SingletonHolder<LogImpl, SingletonWithLongevity> Log;

```

这个方案很容易掌握，也容易理解，虽然它所解决的问题很复杂。

6.12 摘要

本章导入初期，我描述了 Singleton 的 C++ 流行解法。“防止 Singleton 出现多份实体”的技术相对而言很容易，因为这方面有很好的语言层级的支持。最复杂的问题在于管理 Singleton 的生命期（寿命），尤其是其析构动作。

检测“析构之后的访问动作”十分简单而廉价。这种 dead-reference 检测动作应当成为任何 Singleton 实作品的一部分。

我在本章主题上演绎了四个主要变奏：(1) 编译器控制的 Singleton，(2) Phoenix Singleton，(3) 带寿命的 Singleton，(4) 造成“泄漏”的 Singleton。它们每一个各有优缺点。

围绕着 Singleton 设计模式的，还有严肃的线程议题。实现线程安全（thread-safe）的 Singletons 时，“双检测锁定”模式带来很大帮助。

最后本章整理并归纳了上述这些变体——当我们决定定义策略，并根据策略分解 Singletons 时，这些整理和归纳为我们提供了许多帮助。我们为 Singleton 定下三个 policies: Creation、Lifetime 和 ThreadingModel。我们通过四个 template 参数（一个是 client class，另三个是 policies）将这些策略应用到 SingletonHolder class template 中，这些参数涵盖了所有设计组合。

6.13 SingletonHolder Class Template 要点概览

- SingletonHolder 声明如下：

```

template <
    class T,
    template <class> class CreationPolicy = CreateUsingNew,
    template <class> class LifetimePolicy = DefaultLifetime,
    template <class> class ThreadingModel = SingleThreaded
>
class SingletonHolder;

```

- 当你需要具现化 `SingletonHolder` 时，请将你的 `class` 作为第一个 `template` 参数传进去，并组合其他三个参数，用以选择各种设计变化。例如：

```
class MyClass { ... };  
typedef SingletonHolder<MyClass, CreateStatic>  
    MySingleClass;
```

- 你必须定义 `default` 构造函数；或者你必须传进那个常备的（预定义的）`Creation policy` 之外的其他某个 `creator`。
- 三个 `policies` 的封装实作描述于表 6.1。只要有需求，你也可以加上自己的 `policy classes`。

Smart Pointers

智能指针

在全世界程序员和技术作家撰写的无数代码和浩瀚笔墨中，smart pointers（智能指针）已经成为主角之一。也许可以这么说，smart pointers 是最普及、最复杂、最强大的 C++ 工具；其吸引人的地方在于，它涉及语法和语义上的很多议题。本章将讨论 smart pointers：从最简单到最复杂，从实作时会产生的最明显错误到最隐微的错误——其中有些错误极为可憎。

简而言之，smart pointers 是一种 C++ 对象，它提供 `operator->` 和 `unary operator*`，藉以模拟一般指针（simple pointers）的行为。除了提供指针的语法和语义，smart pointers 通常还会在底层完成很多有用的工作（例如内存管理或锁定操作），这么一来，对于 smart pointers 所指对象的生命期，应用程序就不必特意管理。

本章不仅讨论 smart pointers，还实作出一个名为 `SmartPtr` 的 class template。`SmartPtr` 围绕着 policies（见第 1 章）进行设计，成果是一个具有精确安全级别、高效的 smart pointer，并具备你希望获得的易用性。

阅读本章之后，你将在以下 smart pointers 方面成为专家：

- smart pointers 的优点和缺点
- 拥有权管理策略（ownership management strategies）
- 隐式转换（implicit conversions）
- 测试和比较
- 多线程（multithreading）议题

本章实作出一个泛化的、通用的 `SmartPtr` class template。每一小节讨论一个独立的实作议题，最后将所有部分组合在一起。阅读本章，你除了可以理解 `SmartPtr` 的原理，还将学会如何使用它、调整它和扩充它。

7.1 Smart Pointers 基础

什么是 smart pointers？它是一个 C++ class，在语法和某些语义上模拟一般指针，但提供更多功能。由于 smart pointers 指向“类型各异”之对象，所以它们势必拥有大量共通代码，所以现有的高质量 smart pointers 几乎都以 template 完成，template 参数为被指物（pointee）之型别，正

如以下所见：

```
template <class T>
class SmartPtr
{
public:
    explicit SmartPtr(T* pointee) : pointee_(pointee);
    SmartPtr& operator=(const SmartPtr& other);
    ~SmartPtr();
    T& operator*() const {
        ...
        return *pointee_;
    }
    T* operator->() const {
        ...
        return pointee_;
    }
private:
    T* pointee_;
    ...
};
```

`SmartPtr<T>` 将“指向 T 的指针”纳为成员变量 `pointee_`。这是大多数 smart pointers 的做法。某些场合纳入的可能是“数据的 handles”（译注：关于 handles 和 pointers 的不同，可参见 p.161 第 2 段），并通过动态计算获得指针。

上述两个操作符为 `SmartPtr` 带来了类似指针的语法和语义。也就是说，你可以写出这样的程序：

```
class Widget
{
public:
    void Fun();
};

SmartPtr<Widget> sp(new Widget);
sp->Fun();
(*sp).Fun();
```

就 `sp` 而言，除了其定义，没有什么地方显示它不是指针。smart pointers 的好处是：你可以用 smart pointers 来取代指针，代码无需进行大修改。这样，你就可以轻而易举地得到额外好处。如果在大型程序中使用 smart pointers，“将代码的修改量降至最低”这一好处非常吸引人，也非常重要。但是你很可能会看到，smart pointers 不是免费的午餐。

7.2 交易

于是你问了：使用 smart pointers 需要什么交易呢？将一般指针替换为 smart pointers 能得到什么好处？答案很简单。smart pointers 具有“value 语义”，一般指针没有。

对象具有“value 语义”，意指你可以“拷贝 (copy)”这个对象，可以对这个对象“赋值 (assign

to)”。型别 `int` 是高级对象 (first-class object) 的一个绝佳范例。你可以随意产生、拷贝和改变整数的值。用于“遍历缓冲区内部元素”的指针，也具有“value 语义”——初始化时你让它指向缓冲区起始处；然后你移动它，直到缓冲区尾端为止。在这个过程中，为保存中间结果，你可以将它的值（所指对象）拷贝到其他变量中。

但如果指针保存的是“经由 `new` 分配的值”，情况就大不相同。一旦你写出这样的代码：

```
Widget* p = new Widget;
```

变量 `p` 不仅指向分配给 `Widget` 对象的内存，并且拥有 (own) 这一内存。这是因为稍后你必须调用 `delete p`，确保摧毁 `Widget` 对象并释放其内存。如果在这行代码之后的某处，你又写了下面这行语句：

```
p = 0; // assign something else to p
```

那么，你将失去 `p` 原先所指对象的拥有权 (ownership)，并且完全没有机会再次得到它。你造成了资源泄漏；而资源泄漏绝对没有好处。

此外，将 `p` 拷贝至另一个变量时，编译器不会对“指针所指之内存”的拥有权进行自动管理。你得到的将是两个“指向同一对象”的指针，因此你得更加小心地跟踪它们，因为“二次删除”比“不删除”更具灾难性。所以，指针如果指向“因分配而得之对象 (allocated objects)”，将不具备“value 语义”——你不能随意拷贝它，也不能对它赋值。

这种场合下 `smart pointers` 会带来很大帮助。除了提供类似指针的行为，大多数 `smart pointers` 还提供拥有权管理功能。`smart pointers` 可以掌握拥有权的变化情况，它们的析构函数还可以依据定义良好的策略来释放内存。很多 `smart pointers` 都保存着足够的信息，从而可以完全主动地释放它们所指的对象。

`smart pointers` 可以通过各种方式管理拥有权，不同的方式解决不同的问题。某些 `smart pointers` 会自动转移拥有权：在对一个“指向某对象”之 `smart pointer` 进行拷贝之后，“源端”`smart pointer` 变成了 `null`，“目标端”`smart pointer` 则指向该对象（并获得其拥有权）。这正是 C++ *Standard* 的 `std::auto_ptr` 所提供的行为。其他一些 `smart pointers` 采用引用计数 (reference counting)：追踪记录指向同一对象之 `smart pointers` 的总数；一旦总数降为零，就 `delete` 所指对象。此外还有其他 `smart pointers`：无论何时当你拷贝它们时，它们总会复制其所指对象。

简而言之，在 `smart pointers` 的世界中，拥有权 (ownership) 是一项重要课题。经由拥有权管理，`smart pointers` 能够支持“完整性保证”和完整的“value 语义”。拥有权与 `smart pointers` 的构造、拷贝、析构有很大关系，可想而知，这三者将成为 `smart pointer` 最重要的功能。

下面数个小节针对 `smart pointers` 的设计和实作各方面进行讨论。我们的目标是：让 `smart pointers` 尽可能和原始指针 (raw pointers) 相近。这个目标具有矛盾性，毕竟，如果 `smart pointers` 的行为“完全”像原始指针，那它们可不就真的是原始指针了吗☺。

追求 smart pointers 和原始指针之间的兼容性时，“有效提高兼容性”和“步入混乱”之间只有一步之隔。你会发现，有些功能看起来值得加入，但实际上会给客户带来巨大风险。要想实作优良的 smart pointers，就要对它们的功能组合进行细心的选择。

7.3 Smart Pointers 的存储

开始讨论之前，让我问一个有关 smart pointers 的基本问题。pointee_ 的型别必须是 T* 吗？如果不是，它还可以是什么？在泛型程序设计中，你应该总是问自己这样的问题。在一段泛型代码中，每一个写死了的型别都会降低代码的通用性。写死的型别之于泛型代码恰如魔术常量（magic constants）之于一般代码。（译注：所谓魔术常量，指的是直接以“字面值”literal 而非符号所表示的常量）

在某些情况下，允许定制 pointee（被指物）的型别是很有价值的，例如当你处理非标准指针饰词的时候。在 16 bits Intel 80x86 时代，你可以用 `__near`、`__far`、`__huge` 这样的饰词来限定指针。其他分段式内存架构（segmented memory architecture）也使用类似饰词。

另一种有价值的情况是：当你想对 smart pointers 分层（layer）的时候。如果有一个别人设计好的 `LegacySmartPtr<T>` smart pointer，你想对它升级，该怎么办？继承它？那将是个危险的选择。较好的做法是将这个既有的 smart pointer 包装在你自己的 smart pointer 中。这种做法是可能的，因为内部 smart pointer 支持指针语法。从外部 smart pointer 的角度看去，pointee 的型别不是 T*，而是 `LegacySmartPtr<T>`。

smart pointers 分层技术有很多有趣的应用，这主要归因于 `operator->` 机制。当你对某个型别实施 `operator->`，而该型别并非内建指针时，编译器会做一件很有趣的事情：在找出用户自定义之 `operator->` 并将它施行于该（非内建指针）型别后，编译器会对执行结果再次施行 `operator->`。编译器不断执行这样的动作，直至触及一个指向内建型别的指针，然后才进行成员访问。由此可见，smart pointer 的 `operator->` 不一定传回指针，它可以传回一个“实作出 `operator->`”的对象，而且不改变语法。

这导致一种很有趣的技术：“前调用”及“后调用”（pre- and post- function calls）（Stroustrup 2000）。如果你以传值方式从 `operator->` 传回一个 `PointerType` 对象，执行次序将像下面这样：

1. 执行 `PointerType` 的构造函数。
2. 调用 `PointerType::operator->`，传回的很可能是个指针，指向一个 `PointeeType` 对象。
3. 对 `PointeeType` 的成员进行访问动作——很可能是一个函数调用动作。
4. 执行 `PointerType` 的析构函数。

简言之，你可以通过一种巧妙的方法来实现“锁定式函数调用”（locked function call）。这种手法在多线程和资源访问的锁定中有广泛应用。你可以让 `PointerType` 构造函数锁定资源，然后访问资源，最后再由 `PointerType` 析构函数将资源解除锁定。

通用性议题至此尚未结束。有时候，较之 smart pointers 具有的强大资源管理技术，其语法层面的“指针”特性相形之下显得失色，因而在少数特殊情况下 smart pointers 可以放弃指针语法。

“未定义 operator-> 和 operator*”的对象基本上违反了 smart pointers 的根本定义，但即使如此某些这类对象也还是应该像 smart pointers 一样地被对待。

让我们看看现实世界中的 APIs 和应用程序。很多操作系统将 handles 作为某些内部资源（譬如 windows、mutexes、devices）的访问器（accessors）。Handles 是一种被刻意模糊化的指针；它的用途之一是防止用户直接操纵关键性的操作系统资源。通常 handles 是一个整数值，被当做某个隐藏的指针表格（tables of pointers）的索引，这个表格为内部系统和程序员之间提供了一层额外间接性。虽然 handles 并未提供 operator->，但它们在语义和管理方式上很像指针。

对这样一种 “smart resource” 提供 operator-> 或 operator* 并没有意义，但是在它们身上，你的确可以利用 smart pointers 所特有的一切资源管理技术。

为了概括出 smart pointers 国界中的一般性，我们得分清 smart pointer 之中可能存在的三种型别，这三种型别有明显的差异：

- **storage type**。这是 pointee_ 的型别。缺省情况下（亦即在一般的 smart pointers 中）它是一个原始指针（raw pointer）。
- **pointer type**。这是 operator-> 的返回型别。这个型别可以不同于 storage type——如果你想传回一个代理对象（proxy object）而不是一个指针的话。本章稍后你会看到使用代理对象的例子。
- **reference type**。这是 operator* 的返回型别。

如果 SmartPtr 能够以某种灵活方式支持这种一般性，将会十分得益。因此，这里提到的三种型别应该抽象化为一个 policy（策略），我们称之为 Storage。

总言，smart pointers 能够（也应该）将它们的 pointee 型别泛型化。为此，SmartPtr 在 Storage policy 中将上述三种型别抽象化为 stored type、pointer type、reference type。对于某个 SmartPtr 具现实体来说，并非所有型别都一定有意义，少数情况下（例如 handles）某个 policy 可能会禁止取用 operator-> 或 operator*，或两者都禁止。

7.4 Smart Pointer 的成员函数

现有的很多 smart pointer 实作品都允许通过成员函数执行各种操作，例如用于取用 pointee 对象的 Get()，用于修改 pointee 对象的 Set()，用于移转拥有权的 Release()。这是封装 SmartPtr 功能的一种显然而又自然的方式。

但是经验证明：成员函数不大适合 smart pointers。原因是，对 “smart pointer” 成员函数的调用和对 “所指对象” 成员函数的调用，极易混淆。

例如，假设你有一个 Printer class，提供两个成员函数，分别为 Acquire() 和 Release()。通

过 `Acquire()` 你可以获得打印机的拥有权，于是其他应用程序就不会用它来打印；通过 `Release()` 你可以放弃拥有权。当你对着 `Printer` 运用 `smart pointer` 时，你会发现，有两个语义上相差悬殊的东西，在语法上竟有奇怪的类似。

```
SmartPtr<Printer> spRes = ...;
spRes->Acquire(); // acquire the printer
... print a document ...
spRes->Release(); // release the printer
spRes.Release(); // release the pointer to the printer
```

你看，`SmartPtr` 的用户可以访问到两个完全不同的世界：“所指对象的成员”世界和“`smart pointer` 的成员”世界。只有 `dot` 操作符和 `arrow` 操作符勉强分隔了这两个世界。

表面看来，C++ 的确要求你必须习惯性地注意语法上的某些细微差异。如果 Pascal 程序员学习 C++，他们甚至会觉得 `&` 和 `&&` 之间的细微差异十分令人讨厌，但 C++ 程序员早已见多不怪了，他们养成了一种习惯，可以轻易区分这种语法差异。

但是 `smart pointer` 的成员函数打破了 C++ 程序员养成的习惯。原始指针没有成员函数，所以，究竟是 `dot call` 或是 `arrow call`，C++ 程序员的眼睛不习惯去检查和区分。对此，编译器可以做得很好：如果你在一个原始指针后使用了 `dot` 操作符，编译器会报错。所以不难想象（而且经验也已证明）：即使老练的 C++ 程序员也会觉得，如果 `sp.Release()` 和 `sp->Release()` 都能畅通无阻地通过编译，而它们做的又是完全不同的事，那将带来极大麻烦。解决办法很简单：`smart pointers` 不应该使用成员函数。`SmartPtr` 只使用非成员函数，这些函数是 `smart pointer class` 的 “friend”。

和 `smart pointers` 成员函数一样，重载函数（overloaded functions）也会造成混淆，但它有着重要的不同。C++ 程序员总在使用重载函数，那是 C++ 语言重要的组成，并经常用于程序库和应用程序开发中。这意味着在撰写和阅读代码时，C++ 程序员一定会去注意函数调用语法的差异——譬如 `Release(*sp)` 和 `Release(sp)` 之间的差异。

`SmartPtr` 有必要保留的成员函数只有构造函数、析构函数、`operator=`、`operator->` 和 `unary operator*`。`SmartPtr` 的其他所有操作都经由具名的（named）非成员函数提供。

为了代码的清晰，`SmartPtr` 没有提供任何具名的“成员函数”。访问 `pointee` 对象的仅有函数是 `GetImpl`、`GetImplRef`、`Reset` 和 `Release`，它们都定义于命名空间层级。

```
template <class T> T* GetImpl(SmartPtr<T>& sp);
template <class T> T& GetImplRef(SmartPtr<T>& sp);
template <class T> void Reset(SmartPtr<T>& sp, T* source);
template <class T> void Release(SmartPtr<T>& sp, T*& destination);
```

- `GetImpl` 传回 `SmartPtr` 保存的指针对象。
- `GetImplRef` 传回 `SmartPtr` 保存的指针对象的 `reference`。`GetImplRef` 可让你改变底部指针，因此使用时要极为小心。
- `Reset` 将底部指针重新设为另一个值，并释放前一个值。

- Release 释放 smart pointer 的拥有权，让用户负责管理 pointee 对象的生命。

在 Loki 程序库中，这四个函数的声明实际上要略微复杂些。它们并不假设 SmartPtr 保存的指针对象的型别是 T* ——正如 7.3 节所述，指针型别由 Storage policy 来确定。大多数情况下那将是个直接的指针，除非使用的是外来的 Storage 实作品——此时它可能是个 handle，或是某个复杂型别。

7.5 拥有权 (Ownership) 管理策略

smart pointers 之所以存在，“拥有权的管理”往往是最重要的理由。通常，从客户的角度来看，smart pointers 拥有其所指对象。smart pointer 是一种 “first-class value”，它负责在底层删除它所指向的对象。客户可以通过“调用辅助管理函数”来干预 pointee 的寿命。

为了实现“自我拥有” (self-ownership)，smart pointers 必须小心跟踪 pointee 对象，特别是在拷贝、赋值和析构期间。这种跟踪带来了某些额外开销——空间上的、时间上的，或者兼而有之。应用程序应该选用“最适合解决手中问题，同时不会带来太大开销”的策略。

下面数小节讨论的是最普遍的拥有权管理策略，以及 SmartPtr 如何实作这些策略。

7.5.1 深层拷贝 (Deep Copy)

最简单的策略是：只要拷贝 smart pointer，就一定也拷贝 pointee 对象。如果确保这一点，每一个 pointee 对象就只会会有一个 smart pointer，于是 smart pointer 析构函数就可以安全地 delete pointee 对象。采用这种深层拷贝策略的 smart pointers，情况如图 7.1 所示。

乍看之下深层拷贝策略好像很无趣；smart pointer 似乎没有并没有在正规的 C++ “value 语义”之上再添加什么价值。既然简单地“以 by value 方式传递 pointee 对象”也可以解决问题，为什么还要费尽周折地使用 smart pointer 呢？

答案是：为了支持多态 (polymorphism)。smart pointers 是一种安全传送多态对象的工具。你虽然拥有一个指向 base class 的 smart pointer，它可能实际指向某个 derived class。当你拷贝这个 smart pointer 时，你也想拷贝它的多态行为。有趣的是你不一定确切知道你正在处理的是何种行为和状态，但你确实需要复制这种行为和状态。

由于深层拷贝大都和多态对象有关，因此下面这个幼稚的 copy 构造函数是错误的：

```
template <class T>
class SmartPtr
{
public:
    SmartPtr(const SmartPtr& other)
        : pointee_(new T(*other.pointee_))
    {
    }
    ...
};
```

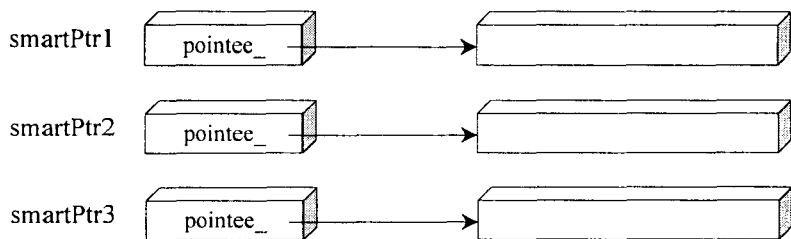


图 7.1 采用急拷贝（eager copy，相对于缓拷贝 lazy copy）之 smart pointer 的内存布局

假设你想拷贝一个型别为 `SmartPtr<Widget>` 的对象。如果 `other` 指向 `ExtendedWidget` class 的一个实体，而 `ExtendedWidget` 从 `Widget` 派生而来，那么上述的 `copy` 构造函数将只会拷贝 `ExtendedWidget` 对象的 `Widget` 部分。这便是所谓的“切割（slicing）”现象——按理说 `ExtendedWidget` 对象会比 `Widget` 对象更大一些，但一旦出现切割现象，只有其 `Widget` 部分会被拷贝。一般来说“切割”并非大家希望看到的现象，遗憾的是 C++ 却允许切割如此容易地发生：一个简单的 `call by value` 就足以切割对象，而且不带任何警告。

第 8 章将就 cloning（复制、克隆）技术作深入探讨。该处讨论告诉我们，对于一个继承体系，如果要想得到多态的 `clone` 行为，典型做法是定义一个 `virtual Clone()`，并像下面这样实作：

```
class AbstractBase
{
    ...
    virtual Base* Clone() = 0;
};

class Concrete : public AbstractBase
{
    ...
    virtual Base* Clone()
    {
        return new Concrete(*this);
    }
};
```

在所有 `derived classes` 中，`Clone()` 的实作必须沿用这一相同模式（pattern）；尽管这会带来重复的结构，但除了这么做之外，我们没有什么合适的方法——除了宏——可以自动定义 `Clone()` 成员函数。

一个泛型 `smart pointers` 不可能知道 `cloning` 成员函数的确切名称——可能是 `clone()`，也可能是 `MakeCopy()`。所以最灵活的方法是：以一个专门负责 `cloning` 事务的 `policy` 来将 `SmartPtr` 参数化。

7.5.2 临写拷贝（Copy on Write）

“临写拷贝”（爱好者昵称为 **COW**）是一种优化技术，其目的是避免非必要的对象拷贝。COW 的根本思想是：在第一次修改 `pointee` 对象时复制之，在此之前多个指针可以共享该对象。

然而 `smart pointers` 并不是实作 COW 的最佳场所，因为 `smart pointers` 无法区分 `pointee` 对象的 `const` 成员函数调用和 `non-const` 成员函数调用。下面是个例子：

```
template <class T>
class SmartPtr
{
public:
    T* operator->() { return pointee_; }
    ...
};

class Foo
{
public:
    void ConstFun() const;
    void NonConstFun();
};

...
SmartPtr<Foo> sp;
sp->ConstFun(); // invokes operator->, then ConstFun
sp->NonConstFun(); // invokes operator->, then NonConstFun
```

两个被调用的函数都调用同一个 `operator->`；因此，对于是否应当执行 COW，上述这个 `smart pointer` 没有任何线索可循。毕竟对 `pointee` 对象的函数调用系发生于“超越 `smart pointer` 范围之外”的某个地方。（7.11 节对于 `const` 和 `smart pointer` 及其所指对象之间的互动有所说明）

总而言之，对在具有完整功能的 `classes` 而言，作为实作上的一种优化，COW 是有效的。但是 `smart pointers` 处于太低层次，无法有效实作 COW 语义。当然，在为 `classes` 实作 COW 时，`smart pointers` 可以成为一个很好的构件（基础材料，`building block`）。

本章实作的 `SmartPtr` 没有支持 COW。

7.5.3 Reference Counting（引用计数）

这是 `smart pointers` 最普遍采用的一种拥有权策略。此法追踪“指向同一对象”的 `smart pointers` 数目。当数目降为零时，`pointee` 对象自动被删除。只要不破坏某些规则（例如不让 `dumb pointers` 和 `smart pointers` 指向相同对象），这个策略非常有效。

实际的计数器必须在 `smart pointer` 对象之间共享，从而形成图 7.2 所示的结构。除了“指向对象本身”的指针外，每个 `smart pointer` 还需保存一个指向 `reference counter`（引用计数器）的指针（亦即图 7.2 的 `pRefCount_`）。这通常会使 `smart pointer` 翻涨一倍大小，这个额外开销是否可

被接受，取决于你的需要和你所受的限制。

还有另一个更隐微的、关于额外开销的问题。reference-counted smart pointers 必须在自由空间（free store）中保存计数器。问题是，在很多实作品中，缺省的 C++ 自由空间分配器在分配小型对象时显得极为低速，并且浪费空间，这一点在第 4 章已有论述。计数器一般占用 4 个 bytes，显然属于小型对象。速度上的开销源于查找“内存可用区块（chunks）”时的低速算法，空间上的开销则源于分配器为每个 chunk 保存的簿记（bookkeeping）信息。

如果将指针和计数器保存在一起，如图 7.3 所示，那么相关空间开销就能有一定程度的减少。如果采用图 7.3 所示的结构，一个 smart pointer 的大小可以降至一个指针大小，但牺牲了访问速度：pointee 对象成为远离于外的额外间接层。这是一个相当大的弊病，因为面对一个 smart pointer，你固然只会构造、摧毁它一次，却往往需要使用它多次。

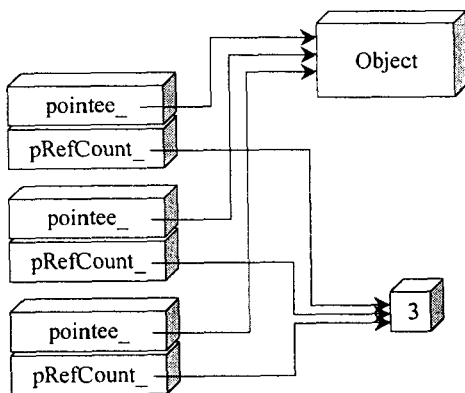


图 7.2 三个 reference-counted smart pointers，指向同一对象

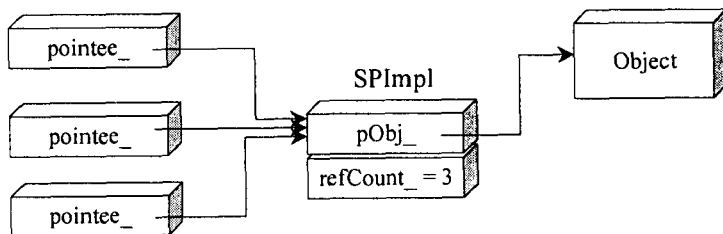


图 7.3 reference-counted pointer 的另一种结构

最有效率的解法是通过 `pointee` 对象本身来保存计数器，如图 7.4 所示。这样一来，`SmartPtr` 将只会是一个指针大小，而且完全没有额外开销。这一技术被称为“侵入式引用计数” (*intrusive reference counting*)，因为在 `pointee` 中计数器是个“侵入者”——从语义上说计数器应该隶属于 `smart pointer`。这个名称也暗示，就像阿契里斯 (Achilles，希腊神话人物) 的脚踵一样，这一技术也有其弱点：要想支持 `reference counting`，你必须先对 `pointee class` 做好相关设计，否则你就得修改既有的 `pointee class`。

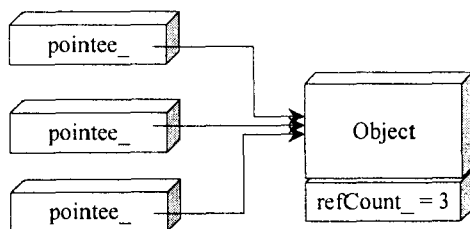


图 7.4 侵入式 (intrusive) 引用计数

作为泛型 `smart pointer`，只要有可能，你都应该使用“侵入式引用计数” (*intrusive reference counting*)，但同时也应该实作“非侵入式引用计数”，作为一种可被接受的选择性方案。实作“非侵入式引用计数”时，第 4 章的小型对象分配器会带来很大帮助。本章的 `SmartPtr` 采用“非侵入式引用计数”，就运用了小型对象分配器，大大降低计数器可能招致的效率额外开销。

7.5.4 Reference Linking (引用链接)

Reference linking 基于一条实际观察到的经验：你并不真的需要确切知道“指向某一 `pointee` 对象”的 `smart pointer` 的实际数目；你只是需要检测那个数目何时降为零。这一经验导致的想法是：维护一个 `ownership list` (拥有权列表)，如图 7.5 所示²⁶。

所有“指向某一 `pointee`”的 `SmartPtr` 对象都形成一个 `doubly linked list` (双向链表)。一旦从现有的 `SmartPtr` 生成新的 `SmartPtr` 时，新对象被附添 (append) 于 `list` 之中；`SmartPtr` 析构函数负责将“被毁对象”从 `list` 中删除。一旦 `list` 为空，`pointee` 对象即被删除。

`doubly linked list` 用于“引用追踪” (*reference tracking*) 极为合适。你不能使用 `singly linked list` (单向链表)，因为从中删除元素将消耗线性时间。你也不能使用 `vector`，因为 `SmartPtr` 对象不是连续的 (况且从 `vectors` 中删除元素也需要线性时间)。你需要一个“添加、删除、空检测三项操作都耗用常数时间”的结构。这一角色责无旁贷地落在 `doubly linked list` 身上。

²⁶ Risto Lankinen 在 1995/09 的 Usenet 中描述了这种 `reference-linking` 机制。

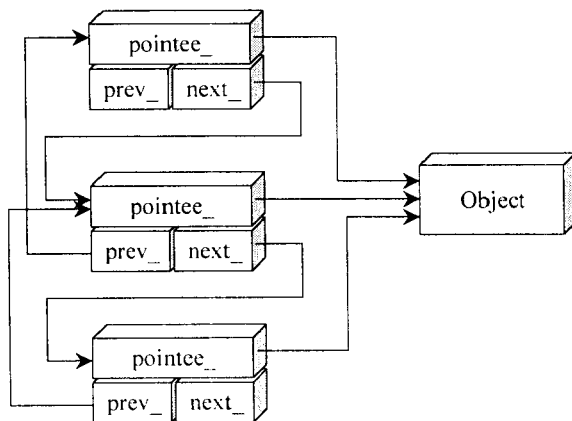


图 7.5 运转中的 reference linking

在 reference-linking 实作中，每个 SmartPtr 对象保存着两个额外指针，一个指向后一元素，另一个指向前一元素。

较之 reference-counting（引用计数），reference-linking（引用链接）的优点在于不花费额外的自由空间，这使它更具可靠性：产生一个 reference-linking smart pointer 必然不会失败。其缺点在于需要更多内存来保存簿记信息（reference-linking 需要三个指针，而 reference-counting 只需要一个指针和一个整数）。此外，reference-counting 的速度快一些，因为拷贝 smart pointers 时只涉及一层间接性和一个递增运算，list 的管理则稍微复杂一些。我们的结论是，只有在自由空间不足的时候才应该使用 reference-linking，否则请优先使用 reference-counting。

结束 reference-counting 管理策略的讨论之前，让我们注意它的一个显著缺点。referenc 的管理——无论是以计数方式还是链接方式——都是所谓 *cyclic reference*（环路引用）资源泄漏的受害者。想象一下：对象 A 拥有一个指向对象 B 的 smart pointer；对象 B 亦拥有一个指向 A 的 smart pointer，这两个对象形成了 *cyclic reference*；即使你不再使用它们之中的任何一个，它们之间却相互引用。reference 管理策略无法检测这种环路，因而这两个对象分配后会永远存在。这样的环路（cycles）有可能跨越多个对象，形成一个封闭圈，常常表现出意想不到的依存性，极难调试。

尽管如此，“reference 管理”是一种强固而高速的拥有权管理策略。如果做好防范措施，“reference 管理”可以为程序开发带来更多便利。

7.5.5 摧毁式拷贝（Destructive Copy）

摧毁式拷贝正如你所想象地那样工作：拷贝期间它会摧毁被拷贝之物。就 smart pointers 而言，摧毁式拷贝是通过“得到源端 smart pointer 的 pointee 对象，然后将它传给目标端 smart pointer”的方式来摧毁源端 smart pointer。std::auto_ptr class template 采用的就是摧毁式拷贝。

“摧毁”二字不仅表达了这一策略的执行行为，还生动表达了它具有的危险性。误用摧毁式拷贝会对程序中的数据、程序的正确性以及你的脑细胞带来毁灭性的后果。

藉由摧毁式拷贝的使用，smart pointers 可以保证任何时刻都只以单一 smart pointer 指向某个对象。一旦将某个 smart pointer “拷贝”或“赋值”给另一个 smart pointer，那个“活动中的”指针会被传递给目标端 smart pointer，源端 smart pointer 的 pointee_ 则设为零。以下代码演示一个采用摧毁式拷贝之简单 SmartPtr 的 copy 构造函数和 assignment 操作符的实作手法。

```
template <class T>
class SmartPtr
{
public:
    SmartPtr(SmartPtr& src)
    {
        pointee_ = src.pointee_;
        src.pointee_ = 0;
    }
    SmartPtr& operator=(SmartPtr& src)
    {
        if (this != &src)
        {
            delete pointee_;
            pointee_ = src.pointee_;
            src.pointee_ = 0;
        }
        return *this;
    }
    ...
};
```

C++ 常规倡议，copy 构造函数和 assignment 操作符右侧应该是 “a reference to a const object”。出于显而易见的原因，采用摧毁式拷贝技术的 classes 违反了这一常规。常规的存在总是有其道理，如果你打算违反它，你就得预估它的负作用。确实如此，这儿就是：

```
void Display(SmartPtr<Something> sp);
...
SmartPtr<Something> sp(new Something);
Display(sp);      // sinks sp
```

虽然 Display 没想伤害其引数（以传值方式传递进来），但它却像一股吞噬 smart pointers 的漩涡，淹没了所有传递给它的 smart pointers。调用 Display(sp) 之后，sp 拥有的是 null 指针。

采用“摧毁式拷贝”的 smart pointers 并不支持“value 语义”，所以不能存储于容器内，而且一般来说，我们几乎得像面对原始指针（raw pointers）那样小心地处理它们。

“将 smart pointers 存储于容器内”的能力十分重要。如果将原始指针（raw pointers）存储于容器内，手工管理拥有权将非常棘手；所以很多用来存储指针的容器可以运用 smart pointers 的优

点。但是采用“摧毁式拷贝”的 smart pointers 不能和容器混合使用。

从正面来说，采用“摧毁式拷贝”的 smart pointers 也有其明显优点：

- 它们几乎不会带来额外开销。
- 它们在厉行“拥有权移转”语义方面表现很好。这种情况下你是在利用前面所说的“漩涡效应”：你清楚地令你的函数接管了传入的指针。
- 它们适合作为函数返回值。如果在 smart pointer 的实作中运用某种技巧²⁷，你可以从函数中传回采用“摧毁式拷贝”的 smart pointer。这么一来，如果调用者未曾使用传回值，你也可以保证 pointee 对象会被摧毁。
- 在拥有多条返回路径的函数中，它们可以非常出色地扮演 stack 变量。你无需记住手工删除某个 pointee 对象，smart pointer 自会为你效劳。

C++ 标准程序库中的 `std::auto_ptr` 便是采用了摧毁式拷贝策略。这为摧毁式拷贝带来另一项重要优点：

- 具有摧毁式拷贝语义的 smart pointers 是 C++ Standard 所提供的唯一一种 smart pointer，这意味着程序员迟早都会习惯它的行为特征。

基于这些原因，SmartPtr 实作品应该支持可供选择的摧毁式拷贝语义。

Smart pointers 可以使用各种拥有权语义（ownership semantics），各有利弊。其中最重要的技术是 **deep copy**（深层拷贝）、**reference counting**（引用计数）、**reference linking**（引用链接）和 **destructive copy**（摧毁式拷贝）。藉由 Ownership policy，SmartPtr 实作出所有这些策略，用户可以选择最适用的一个来满足程序需要。缺省策略是 reference counting。

7.6 Address-Of（取址）操作符

为了使 smart pointers 和其 dumb pointers 尽可能相似，设计者做了艰苦的努力；在这一过程中他们意外发现，`unary operator&`²⁸，亦即 address-of（取址）操作符，似乎可以重载。

Smart pointers 的实作者可能打算这样重载 address-of 操作符：

```
template <class T>
class SmartPtr
{
public:
```

²⁷ 这一技巧由 Greg Colvin 和 Bill Gibbons 首创，用于 `std::auto_ptr`。

²⁸ 之所以称为 `unary operator&`，是为了和 `binary operator&`（亦即 bitwise AND）操作符区分开来。

```
T** operator&()  
{  
    return &pointee_;  
}  
...  
};
```

毕竟，如果 `smart pointer` 打算模拟指针，它的地址就必须可以和一般指针的地址互相替代。有了这份重载，下面这样的代码就有可能成立了：

```
void Fun(Widget** pWidget);  
...  
SmartPtr<Widget> spWidget(...);  
Fun(&spWidget); // okay, invokes operator* and obtains a  
                // pointer to pointer to Widget
```

`smart pointers` 和其 `dumb pointers` 有如此精确的兼容性，看上去似乎很不错，但实际上，`unary operator&` 的重载是弊大于利的花哨技术之一。它之所以不是个好主意，出于以下两个原因。

第一个原因是：暴露了所指对象的地址，这也就意味着完全放弃了对拥有权的自动管理。当客户可以自由访问原始指针的地址时，`smart pointer` 拥有的一切辅助结构（例如 `reference counts`）便在许多用途中失去了效力。如果客户直接处理那个原始指针地址，则 `smart pointer` 一无所知。

第二个原因更具实际意义：`unary operator&` 的重载使得 `smart pointer` 无法用于 STL 容器。事实上某个型别如果重载了 `unary operator&`，该型别就几乎不可能参与泛型编程，因为一个对象的地址是太基本的特性了，绝不能对它儿戏。大多数泛型代码都假设，如果对型别 `T` 的对象施行 `&` 操作符，应该传回 `T*` 对象，可见取址是个基本概念。如果你藐视这一概念，那么泛型代码要么在编译期表现异常，要么（更严重地）就在执行期出糗。

因此，为 `smart pointers` 或任何一般对象重载 `operator&`，都是不值得提倡的。`SmartPtr` 并没有重载 `unary operator&`。

7.7 隐式转换 (Implicit Conversion) 至原始指针型别

看看这段代码：

```
void Fun(Something* p);  
...  
SmartPtr<Something> sp(new Something);  
Fun(sp); // OK or error?
```

这段代码应该通过编译吗？顺应“最大兼容性”的思路，答案是：应该。

从技术上来说，要让前面那段代码通过编译是很简单的，我们只需引入一个用户自定义转换式（user-defined conversion）：

```
template <class T>
class SmartPtr
{
public:
    operator T*() // user-defined conversion to T*
    {
        return pointee_;
    }
    ...
};
```

但是故事并未就此结束。

在 C++ 中，用户自定义转换式（user-defined conversion）有一段有趣的历史。回溯 20 世纪 80 年代，用户自定义转换式刚被引入的时候，大多数程序员都认为那是一项伟大的发明。用户自定义转换式有望带来更统一的型别系统、更具表达力的语义，以及定义“和内建型别毫无区别”之新型别的能力。但是随着时间推移，用户自定义转换式暴露出自身的笨拙和潜在的危险。特别是如果它们暴露了内部数据的 handle（Meyers 1998a，条款 29），它们会变得很危险，而上页代码中的 `operator T*` 正属于这种情况。为自己设计的 smart pointers 提供自动转换式之前，需要三思，原因也正在于此。

如果允许用户随意访问 smart pointer 包装起来的原始指针，必将带来潜在危险。传递原始指针会破坏 smart pointer 的内部工作方式。一旦脱离包装层的束缚，原始指针便能够再度轻易地对程序健全性造成威胁，就像未曾引入 smart pointers 一样。

另一个危险是，用户自定义转换式会在意想不到的情况下冒出来运作——即使当时你不需要它们。请看以下代码：

```
SmartPtr<Something> sp;
...
// A gross semantic error
// However, it goes undetected at compile time
delete sp;
```

编译器会将“delete 操作符”拿来和(转型为 `T*` 的)用户自定义转换式匹配。运行时期，`operator T*` 先被调用，然后 `delete` 作用于其调用结果。你当然不想对 smart pointer 做这样的事情，因为 smart pointer 本可自我管理拥有权。由于多出这个意料之外的 `delete` 动作，smart pointer 底层精心提供的“拥有权管理”都被抛出了九霄云外。

有不少方法可以防止这种 `delete` 动作通过编译，某些方法很有创意（Meyers 1996）。其中有一种方法实作起来十分简单有效：刻意令 `delete` 带有“歧义性（ambiguous）”。做法是提供“两个”自动转换函数，转换后的型别都可用于 `delete`。这两个型别一个是 `T*` 本身，另一个可以是 `void*`。

```
template <class T>
class SmartPtr
{
public:
    operator T*() // User-defined conversion to T*
    {
        return pointee_;
    }
    operator void*() // Added conversion to void*
    {
        return pointee_;
    }
    ...
};
```

对这样一个 **smart pointer** 对象进行 `delete` 动作，将带来歧义性（模棱两可）。编译器无法确定该使用哪一个转换式；上述技巧良好地运用了这种不确定性。

不要忘记，“禁止 `delete`”只是本问题的一部分。实作 **smart pointer** 时，是否提供“自动转换至原始指针”，依然是一个重要决策。它太危险了，以致于我们无法接受它；但它又太方便了，以致于我们无法拒绝它。为此，**SmartPtr** 的做法是给你一个选择。

然而，禁止隐式转换并非意味着要消除对原始指针的所有访问；有些时候那种访问是必要的。因此所有 **smart pointers** 都通过函数调用，提供对“其所包装之指针”的显式取用，做法是：

```
void Fun(Something* p);
...
SmartPtr<Something> sp;
Fun(GetImpl(sp)); // OK, explicit conversion always allowed
```

“能否访问被包装之指针”其实不是问题，问题在于访问时的难易程度。这好像只是微小差异，其实非常重要。隐式转换是在程序员或维护者未注意甚至不知道的情况下发生的。显式转换（就像上述调用 `GetImpl` 一样）则是经过程序员的大脑、思考和手指，并且是白纸黑字，人人可见。

smart pointer 至 **raw pointer**（原始指针）的隐式转型的确不错，但有时很危险。**SmartPtr** 将这种隐式转型视为一种选择，由你决定。缺省采用的是其较为安全的一面：无隐式转换，但通过 `GetImpl()` 函数你总是能够执行显式访问。

7.8 相等性 (Equality) 和不等性 (Inequality)

C++ 告诉其用户：任何如上一节演示的那种技巧（刻意歧义），都会产生一个新环境（context），这一环境可能反过来造成意想不到的影响。

请考虑 smart pointers 的“相等性”和“不等性”测试问题。原始指针（raw pointer）支持的每一种比较动作（comparison），smart pointer 也应该同样予以支持。程序员希望以下测试能够通过编译，并能够像原始指针那样地运作。

```
SmartPtr<Something> sp1, sp2;
Something* p;
...
if (sp1)           // Test 1: direct test for non-null pointer
...
if (!sp1)          // Test 2: direct test for null pointer
...
if (sp1 == 0)      // Test 3: explicit test for null pointer
...
if (sp1 == sp2)    // Test 4: comparison of two smart pointers
...
if (sp1 == p)      // Test 5: comparison with a raw pointer
...

```

如果考虑对称性和 operator!=，将会有更多测试，比这里演示的还要多。只要解决相等性测试，我们便可以轻易定义出相应的对称形式以及不等性测试。

遗憾的是，前面那个问题的解决方案（防止 delete 通过编译）和本问题的一个可能解决方案之间存在冲突。如果只有一个“转向 pointee 型别”的用户自定义转换式，上述大多数测试句（第 4 个除外）都能通过编译，并能够如预期般工作。不利的方面是，你可能会意外地对 smart pointer 调用 delete 操作符。如果有了“两个”用户自定义转换式（刻意造成出来的歧义），你可以检测出不正当的 delete，但上面列出的那些测试也都无法通过编译，因为它们也都带有了歧义。

如果增加一个“转型至 bool”的用户自定义转换式，会有帮助；但不出大家意料，这又带来了新麻烦。假设有这样一个 smart pointer：

```
template <class T>
class SmartPtr
{
public:
    operator bool() const {
        return pointee_ != 0;
    }
    ...
};

```

前 4 个测试可通过编译，但下面 4 个荒唐的操作竟然也通过编译：

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2; // Orange is unrelated to Apple
if (sp1 == sp2)        // Converts both pointers to bool
                        // and compares results
...
if (sp1 != sp2)        // Ditto
...

```



```

bool b = spl;           // The conversion allows this, too
if (spl * 5 == 200)     // Ouch! SmartPtr behaves like an integral
                        // type!
    ...

```

看得出来，我们的做法要么不行，要么过火：一旦增加了一个“转型至 `bool`”的用户自定义转换式，`SmartPtr` 就会在很多地方（多得超过你的实际需要）表现得像个 `bool`。因此，为 `smart pointer` 定义 `operator bool` 不是个明智解法。

在这种进退两难的情况下，正确、完整、可靠的解法是：采取完全一致的做法，分别为每一个操作符定义一份重载版本。这样一来，对一般指针有意义的任何操作，也都会对 `smart pointer` 有意义，并且不会有其他多余操作。以下程序片段完成了这一想法。

```

template <class T>
class SmartPtr
{
public:
    bool operator!() const // Enables "if (!sp) ..."
    {
        return pointee_ == 0;
    }
    inline friend bool operator==(const SmartPtr& lhs,
                                  const T* rhs)
    {
        return lhs.pointee_ == rhs;
    }
    inline friend bool operator==(const T* lhs,
                                  const SmartPtr& rhs)
    {
        return lhs == rhs.pointee_;
    }
    inline friend bool operator!=(const SmartPtr& lhs,
                                  const T* rhs)
    {
        return lhs.pointee_ != rhs;
    }
    inline friend bool operator!=(const T* lhs,
                                  const SmartPtr& rhs)
    {
        return lhs != rhs.pointee_;
    }
    ...
};

```

是的，这样做很痛苦，但它解决了几乎所有“比较”问题，包括对字面值“零”的测试。在上述代码中，那些转发操作符 (forwarding operators) 所做的是：将客户代码中“作用于 `smart pointer` 身上”的操作符传递到 `smart pointer` 所包装的原始指针 (raw pointer) 身上。没有任何模拟能够比这个更逼真现实了。

但我们还没有完全解决问题。如果提供一个“转型至 `pointee` 型别”的自动转换式，还是会有歧义风险。假设你有一个 `Base` class，以及一个从 `Base` 继承而来的 `Derived` class，那么，下面的代码是有实际意义的，但由于存在歧义（模棱两可，*ambiguity*），它却表现不佳。

```
SmartPtr<Base> sp;
Derived* p;
...
if (sp == p) {} // error! Ambiguity between:
                // '(Base*)sp == (Base*)p'
                // and 'operator==(sp, (Base*)p)'
```

的确，`smart pointer` 的开发不适合胆小鬼。

但我们并不是没有子弹了。除了定义 `operator==` 和 `operator!=` 之外，我们还可以增加它们的 `templated` 版本，如下所示：

```
template <class T>
class SmartPtr
{
public:
    ... as above ...
    template <class U>
    inline friend bool operator==(const SmartPtr& lhs, const U* rhs)
    {
        return lhs.pointee_ == rhs;
    }
    template <class U>
    inline friend bool operator==(const U* lhs, const SmartPtr& rhs)
    {
        return lhs == rhs.pointee_;
    }
    ... similarly defined operator!= ...
};
```

从某种意义上来说，这些 `templated` 操作符很“贪婪”，因为它们可以将“比较操作”拿来和任何指针型别匹配（*match*），从而消除歧义。

既然如此，为什么还要保留 `non-templated` 操作符（亦即接受 `pointee` 型别的操作符）呢？它们永远不会有机会匹配，不是吗（因为 `template` 会匹配任何指针型别，包括 `pointee` 型别本身）？

啊，有一条经验准则是这样的：“永远不”实际上是“几乎永远不”。这条准则在这里也适用。在 `if (sp == 0)` 测试中，编译器尝试以下匹配（*matches*）：

- `templated` 操作符。它们不会匹配，因为 `0` 不是指针型别。字面常数 `0` 的确可以隐式转型为指针型别，但“`templated` 匹配”不含转型。
- `non-templated` 操作符。排除了 `templated` 操作符后，编译器开始尝试 `non-templated` 操作符。经由“从 `0` 到 `pointee` 型别”的隐式转换，其中一个操作符会匹配成功。如果 `non-templated` 操作符不存在，这个测试会产生编译错误。

结论是，“non-templated 比较操作符”和“templated 比较操作符”必须同时存在。

现在来看看，如果比较的是两个“通过不同型别而具现化”的 `SmartPtr`，会发生什么事。

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)
...
```

由于存在歧义，编译器会在这个比较测试上遇到麻烦：这两个 `SmartPtr` 具现体都各自定义了 `operator==`，编译器不知道该选择其中哪一个。我们可以定义一种“歧义消除器 (ambiguity buster)”来避开这个问题，如下所示：

```
template <class T>
class SmartPtr
{
public:
    // Ambiguity buster
    template <class U>
    bool operator==(const SmartPtr<U>& rhs) const
    {
        return pointee_ == rhs.pointee_;
    }
    // Similarly for operator!=
    ...
};
```

这个新增的操作符是 `SmartPtr` 的成员之一，专门用来比较 `SmartPtr<...>` 对象。这个歧义消除器的妙处在于，它让 `smart pointers` 之间的比较就像 `raw pointers` 之间的比较一样。如果比较“指向 `Apple`”和“指向 `Orange`”的两个 `smart pointers`，代码本质上就像比较“指向 `Apple`”和“指向 `Orange`”的两个 `raw pointers` 一样。如果这样的比较有意义，代码会通过编译；否则会产生编译期错误。

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> p2;
if (sp1 == sp2) // Semantically equivalent to
    // sp1.pointee_ == sp2.pointee_
...
```

这里遗留了一个无法令人满意的语法问题，也就是 `if (sp)` 这种直接测试。这里的情况变得很有趣。`if` 语句只适用于算术表达式或指针型别表达式，因此，为了让 `if (sp)` 通过编译，我们必须定义一个“转型至算术型别或指针型别”的自动转换式。

我们不提倡“转换至算术型别”，在先前的 `operator bool` 经验上我们已经见证了这一点。指针不是算术型别。是的，就这样，再没什么好说的。“转换至指针型别”合理得多，但问题是这里出现了分歧。

如果想提供“自动转换至 `pointee` 型别”（见前一节），你有两个选择：要么你得承担无意中调

用 `delete` 操作符的风险, 要么你得放弃 `if (sp)` 测试。你只能在“不便”和“冒险”之中作抉择。最后胜利者是“安全”。所以, 你不能写出 `if (sp)` 这样的语句, 但你可以写出 `if (sp != 0)` 或更古怪的 `if (!!sp)`。故事到此结束!

如果不需要提供“转至 `pointee` 型别”的自动转换, 你可以使用一个很有趣的技巧, 让 `if (sp)` 成为可能。在 `SmartPtr` class template 内部, 我们定义一个 inner class `Tester` 和一个“转至 `Tester*`”的转换函数如下:

```
template <class T>
class SmartPtr
{
    class Tester
    {
        void operator delete(void*);
    };
public:
    operator Tester*() const
    {
        if (!pointee_) return 0;
        static Tester test;
        return &test;
    }
    ...
};
```

现在, 如果写出 `if (sp)`, `operator Tester*` 会投入运转。只在 `pointee_` 为 `null` 时这个操作符才传回 `null`。`Tester` 本身禁止 `delete` 操作符, 所以如果有人调用 `delete sp`, 将会出现编译期错误。有趣的是 `Tester` 的定义位于 `SmartPtr` 的 `private` 区, 所以客户代码无法对它进行其他任何操作。

`SmartPtr` 以如下方式处理“相等”和“不等”测试问题:

- 定义两种形式 (templated 和 non-templated) 的 `operator==` 和 `operator!=`。
- 定义 `operator!`。
- 如果允许自动转换至 `pointee` 型别, 那么就额外定义一个转至 `void*` 的转换函数, 刻意令 `delete` call 歧义化。否则定义一个名为 `Tester` 的 `private` inner class, 其中声明 `private operator delete`; 此外还为 `SmartPtr` 定义一个“转至 `Tester*`”的转换式, 并且只有当 `pointee` 对象为 `null` 时, 这个转换函数才传回 `null` 指针。

7.9 次序比较 (Ordering Comparisons)

所谓次序比较操作符, 指的是 `operator<`, `operator<=`, `operator>`, `operator>=`。你可以通过 `operator<` 实作出以上全部操作符。

“是否允许 `smart pointer` 排序”这一问题除了本身带有趣味之外, 还和指针的双重性有关, 而

这种双重性时常给程序员带来困惑。指针集两个概念于一身：迭代器 (iterators) 和代名 (monikers)。指针的迭代特性让你可以经由指针遍历一个 array。指针算术运算 (包括比较运算) 都支持指针的这种迭代特性。此外指针又是代名 (monikers)，一种廉价的对象代表，可以快速行进，并能瞬间访问对象。dereference (提领) 操作符 * 和 -> 都支持代名概念。

指针的这两个特性有时会造成混乱，特别是当你只需要其中之一的时候。如果操作 vector，你会同时使用迭代功能和提领功能，但在遍历 list 或操纵单一对象时，你只使用提领功能。

只有当指针属于同一块连续内存时，我们才会定义指针的次序比较式。换句话说，只有当指针指向“位于同一个 array”中的元素时，你才能对它使用次序比较。

定义 smart pointers 的“次序比较”，可以归结为这样一个问题：让 smart pointers 指向同一 array 中的元素有无意义？从表面判断，答案是 no。Smart pointers 的主要功能是管理对象拥有权，而具有独立拥有权的对象通常不属于同一 array。因而，允许用户进行不合理的比较是危险的。

如果真的需要进行次序比较，你可以经由“显式取用原始指针”来达到。这里的要点在于如何找到一个适用于大多数情况下的最安全、最具表现力的行为。

前一节有过结论：“隐式转换至原始指针”是一项选择性功能。如果 SmartPtr 的客户允许隐式转换，下面的代码就可以通过编译：

```
SmartPtr<Something> sp1, sp2;
if (sp1 < sp2)    // Converts sp1 and sp2 to raw pointer type,
                  // then performs the comparison
    ...
```

这意味着如果想禁止“次序比较”，我们必须先发制人，明白禁止它们。一种做法是：声明它们但不提供定义：这样一来只要你使用它们，就会引发连接错误 (linkage-time error)。

```
template <class T>
class SmartPtr
{ ... };
template <class T, class U>
bool operator<(const SmartPtr<T>&, const U&); // Not defined
template <class T, class U>
bool operator<(const T&, const SmartPtr<U>&); // Not defined
```

更聪明的做法是利用 operator< 来定义其他所有比较操作符。这么一来，如果 SmartPtr 的用户认为最好还是引入 smart pointer 的排序功能，他们只需定义 operator<。

```
// Ambiguity buster
template <class T, class U>
bool operator<(const SmartPtr<T>& lhs, const SmartPtr<U>& rhs)
{
    return lhs < GetImpl(rhs);
}
```

```
// All other operators
template <class T, class U>
bool operator>(SmartPtr<T>& lhs, const U& rhs)
{
    return rhs < lhs;
}
... similarly for the other operators ...
```

请再次注意“歧义消除器”的存在。现在，如果某个程序库用户认为 `SmartPtr<Something>` 应该排序，他只需提供以下代码：

```
inline bool operator<(const SmartPtr<Widget>& lhs,
    const Widget& rhs)
{
    return GetImpl(lhs) < rhs;
}

inline bool operator<(const Widget& lhs,
    const SmartPtr<Widget>& rhs)
{
    return lhs < GetImpl(rhs);
}
```

可惜的是，用户必须定义两个（而非一个）操作符，但这总比定义八个操作符要好得多。

以下将对排序做一个结束，但却是一些不很有趣的细节。有时候，对任意分布（而不只是位于同一 `array`）的对象进行排序很有用处。例如你可能需要存储每个对象的辅助信息，还需要快速访问这些信息。对于这一任务，使用“按对象地址排序”的 `map` 会很有效率。

C++ *Standard* 有助于实作这种设计。虽然 C++ *Standard* 并没有为任意分布的对象定义出指针比较式，但它保证，对于任意两个型别相同的指针，`std::less` 都会产生有意义的结果。因为标准关联式容器（associative containers）以 `std::less` 作为缺省的排序准则，所以你可以安全使用那些以指针为 *keys* 的 `maps`。

`SmartPtr` 也应当支持这一手法，因此 `SmartPtr` 特化了 `std::less`，也就是将调用转发给“用于一般指针”的 `std::less`：

```
namespace std
{
    template <class T>
    struct less<SmartPtr<T> >
        : public binary_function<SmartPtr<T>, SmartPtr<T>, bool>
    {
        bool operator()(const SmartPtr<T>& lhs,
            const SmartPtr<T>& rhs) const
        {
            return less<T*>()(GetImpl(lhs), GetImpl(rhs));
        }
    };
}
```

总之, `SmartPtr` 没有预先定义“次序比较操作符” (ordering operators)。它声明 (但未实作) 两个泛型 `operator<`, 并利用 `operator<` 实作了其他所有“次序比较操作符”。用户可以定义 `operator<` 的特化版本, 或是泛化版本。

藉由 `std::less` 的特化, `SmartPtr` 提供了“对任意 smart pointer 对象进行排序”的支持。

7.10 检测及错误报告 (Checking and Error Reporting)

应用程序对 smart pointers 的安全需求不一而足。某些程序属于“计算密集”, 因而需要对速度进行优化; 另一些 (实际上是大多数) 程序则是“I/O 密集”, 因而允许执行更多执行期检测, 而不至于降低效率。

很多时候, 在同一个应用程序中你可能需要两种模型: 关键地点你需要“低安/高速”; 其他地点你需要“高安/低速”。

我们可以将 smart pointers 的检测问题划分为两类: 初始化检测 (initialization checking) 和提前检测 (checking before dereference)。

7.10.1 初始化检测 (initialization checking)

smart pointers 应该接受 null 吗?

我们可以轻易完成一个保证, 保证 smart pointer 不能为 null: 在实际世界中这非常有用。它意味着任何 smart pointers 总是有效的 (除非你通过 `GetImplRef()` 恣意操作原始指针)。实作上很容易, 我们只需借助一个构造函数: 如果向这个构造函数传入一个 null 指针, 构造函数就抛出一个异常。

```
template <class T>
class SmartPtr
{
public:
    SmartPtr(T* p) : pointee_(p)
    {
        if (!p) throw NullPointerException();
    }
    ...
};
```

但是, 另一方面, null 是保存“非有效指针”的便利贴, 可能常常被运用。

“是否允许 null”也影响着 default 构造函数。如果 smart pointer 不接受 null 值, default 构造函数要如何初始化原始指针呢? 你可以不要 default 构造函数, 但那会使 smart pointers 更难处理。例如, 如果你有一个 `SmartPtr` 成员变量, 但在构造时期没有一个合适的初值, 此时你该怎么办? 因此我们的结论是, 如果想定制初始化行为 (customizing initialization), 就要提供一个合适的缺省值。

7.10.2 提领前检测（checking before dereference）

提领前检测很重要，因为“提领 null 指针”会造成不确定行为（undefined behavior）。对很多应用程序来说，不确定行为是无法让人接受的，所以提领之前对指针进行“合法性检测”是必需的。提领前检测是 SmartPtr 的 operator-> 和 unary operator* 的职责。

和“初始化检测”相比，“提领前检测”可能会成为程序中主要的效率瓶颈，因为在典型的应用程序中，使用（或说提领）smart pointers 比生成 smart pointers 要频繁得多。因此，你必须在安全和速度两方面保持平衡。一条不错的经验法则是：一开始要使用经过严格检测的指针，选定后的 smart pointers 则免除检测，以此来体现对检测的需要。

“初始化检测”和“提领前检测”在概念上能分开吗？不能，因为它们之间有关联。如果初始化阶段作了严格检测，提领前检测就是多余的——因为指针永远有效。

7.10.3 错误报告

对错误提出报告时，唯一合理的选择是抛出一个异常（exception）。

你可以做点事情避免错误发生。例如，如果提领时指针为 null，你可以动态对它初始化。这是一种合法而有价值的策略，称为缓式初始化（lazy initialization）——只在第一次需要其值的时候我们才去构造它。

如果只想在调试期作检测，你可以使用标准的 assert，或类似的更复杂的宏。在发行模式（release mode）中，编译器会忽略这些测试，所以假设你在调试期消除了所有 null 指针错误，你就可以同时获得检测和速度上的好处。

SmartPtr 将检测移交给一个专门的 Checking policy。这个 policy 实作了检测功能（并提供缓式初始化作为选项）和报错策略。

7.11 Smart Pointers to const 和 const Smart Pointers

原始指针允许两种常数性（constness）：“被指对象”的常数性和“指针本身”的常数性。以下对这两种属性举例说明：

```
const Something* pc = new Something;           // points to const object
pc->ConstMemberFunction();                     // ok
pc->NonConstMemberFunction();                  // error
delete pc;                                     // ok (surprisingly)29
Something* const cp = new Something;           // const pointer
cp->NonConstMemberFunction();                  // ok
cp = new Something;                           // error, can't assign to const pointer
const Something* const cpc = new Something;    // const, points to const
```

²⁹ “为什么可以将 delete 操作符施行于 pointers to const 身上”的这类提问，时常会在 comp.std.c++ 新闻群组上引起唇枪舌战。事实上，不论好坏与否，C++ 允许你这么做。


```
cpc->ConstMemberFunction();           // ok
cpc->NonConstMemberFunction();        // error
cpc = new Something;                  // error, can't assign to const pointer
```

相应地，`SmartPtr` 的使用如下：

```
// Smart pointer to const object
SmartPtr<const Something> spc(new Something);
// const smart pointer
const SmartPtr<Something> scp(new Something);
// const smart pointer to const object
const SmartPtr<const Something> scpc(new Something);
```

`SmartPtr` class template 可以检测被指对象的常数性——要么通过偏特化 (partial specialization)，要么使用第 2 章定义的 `TypeTraits` template。后者更可取，因为它不像偏特化那样造成代码重复。

`SmartPtr` 模拟了 `pointers to const` 语义、`const pointers` 语义，以及二者的结合。

7.12 Arrays

大多数场合你都应该避免和“分配于 heap 之上的 arrays”打交道；你应该避免使用 `new[]` 和 `delete[]`；取而代之的是，你最好使用 `std::vector`。`C++ Standard` 提供的 `std::vector` class template 提供了“动态 array”所能提供的一切功能，而且还多得多；此外，在大多数情况下，它造成的额外开销也是可以忽略不计的。

但是，“大多数情况”并不表示“任何情况”。很多情况下你不需要也不希望拥有一个全功能的 `vector`；你需要的只是一个动态分配的 array。这种情况下如果没能让 `smart pointer` 的能力彰显出来，可就有点笨拙了。复杂的 `std::vector` 和动态分配的 array 之间有某种缺口；如果用户需要，`smart pointers` 可以提供 array 语义来填补这一缺口。

如果 `smart pointer` 指向的是 array，那么从它的角度来看，仅有的一个重点是“记得在析构函数中调用 `delete[]` `pointee_` 而非 `delete pointee_`”。`Ownership policy` 已经解决了这问题。

第二个问题是为 `smart pointers` 重载 `operator[]`，从而提供下标访问功能。从技术上说这是可行的；事实上 `SmartPtr` 的早期版本的确提供了一个独立的 `policy`，用以实作可选的 array 语义。但是只在非常少的情况下 `smart pointers` 才指向 array。在那些情况下，如果使用 `GetImpl()`，你就已经有了下标访问：

```
SmartPtr<Widget> sp = ...;
// Access the sixth element pointed to by sp
Widget& obj = GetImpl(sp)[5];
```

唔，为提供语法上的额外便利而引入一个新的 `policy`，似乎不是个好决定。

通过 `Ownership policy`，`SmartPtr` 支持“析构过程的定制”，所以您可以通过 `delete[]` 来处理 array 相关析构。但是请注意，`SmartPtr` 并没有提供指针算术运算。

7.13 Smart Pointers 和多线程（Multithreading）

通常 smart pointers 有助于对象共享，而多线程则影响对象的共享。因此，多线程问题影响 smart pointers。

smart pointers 和多线程之间的影响发生于两个层次。一个是 pointee 对象层，一个是簿记数据层。

7.13.1 Pointee 对象层（Object Level）上的多线程

如果多个线程访问同一对象，而且如果是通过 smart pointer 来访问该对象，那么在 `operator->` 调用期间，锁定这个对象很有必要。这是有可能做到的：你可以让 smart pointer 传回一个 proxy 对象，而非原始指针（raw pointer）。proxy 对象的构造函数锁定 pointee 对象，析构函数则为之解锁。这一技术在 Stroustrup（2000）中有所说明。下面提供的代码演示了这一解法。

首先，让我们看看这样一个 widget class，它有两个锁定用基本工具（locking primitives）：`Lock()` 和 `Unlock()`。调用了 `Lock()` 之后，你可以安全访问对象；其他任何试图调用 `Lock()` 的线程都将被阻塞（block）。调用 `Unlock()` 之后，其他线程才有可能锁定这个对象。

```
class Widget
{
    ...
    void Lock();
    void Unlock();
};
```

接着，我们定义一个 `LockingProxy` class template。其任务是在 `LockingProxy` 生命期内（通过 `Lock/Unlock` 惯常手法）锁定一个对象。

```
template <class T>
class LockingProxy
{
public:
    LockingProxy(T* pObj) : pointee_ (pObj)
    { pointee_->Lock(); }
    ~LockingProxy()
    { pointee_->Unlock(); }
    T* operator->() const
    { return pointee_; }
private:
    LockingProxy& operator=(const LockingProxy&);
    T* pointee_;
};
```

除了构造函数和析构函数外，`LockingProxy` 还定义了 `operator->`，它传回一个 pointer to pointee object。

虽然 `LockingProxy` 看起来有点像 smart pointer, 但它还多了一层: `SmartPtr` class template 本身。

```
template <class T>
class SmartPtr
{
    ...
    LockingProxy<T> operator->() const
    { return LockingProxy<T>(pointee_); }
private:
    T* pointee_;
};
```

请回忆 7.3 节, 那儿曾经说明 `operator->` 机制: 编译器可以对一个 `->` 表达式多次应用 `operator->`, 直到找到原始指针 (raw pointer)。现在假设你发出下面这样一个调用 (假设 `Widget` 定义了 `DoSomething()`) :

```
SmartPtr<Widget> sp = ...;
sp->DoSomething();
```

这里的技巧是: `SmartPtr` 的 `operator->` 传回一个 `LockingProxy<T>` 暂时对象。编译器继续施行 `operator->`。 `LockingProxy<T>` 的 `operator->` 传回 `Widget*`。编译器通过这个 pointer to `Widget` 调用 `DoSomething()`。调用期间, `LockingProxy<T>` 暂时对象处于活跃状态并锁定 `Widget` 对象; 也就是说对象被安全锁定了。一旦 `DoSomething()` 返回, `LockingProxy<T>` 暂时对象被摧毁, `Widget` 对象也随之解锁。

自动锁定 (automatic locking) 是 smart pointer 分层技术 (layering) 的一个良好应用。你可以改变 `Storage policy`, 以同样的方式对 smart pointers 进行分层。

7.13.2 簿记数据层 (Bookkeeping Data Level) 上的多线程

有时候, smart pointers 还会处理 “pointee 对象之外” 的数据。我们在 7.5 节已经看到, reference counted smart pointers 在底层共享了所谓的 “引用计数”。如果你将 reference counted smart pointers 从一个线程拷贝到另一个线程, 你会得到两个 smart pointers, 指向同一个引用计数值。当然, 它们也指向同一个 pointee 对象, 那对用户而言是可访问的, 他们可以锁定它。与此对比的是, 引用计数值对用户而言是不可访问的, 所以引用计数器的管理完全由 smart pointer 负责。

不仅 reference counted pointers 面临多线程方面的危险, 那些在内部保存 “互指彼此” 之指针 (也可算是数据共享) 的 reference-tracked smart pointers (7.5.4 节) 也如此。reference linking 产生了一个 smart pointers 群体 (communities), 但群体中的所有成员并不一定都属于同一个线程。因此, 每次对 reference-tracked smart pointer 执行拷贝、赋值、摧毁动作时, 你必须施以合适的锁定, 否则 doubly linked list 可能会遭受破坏。

总而言之,多线程问题最终会影响到 smart pointers 的实作。让我们看看,如何在 reference counting 和 reference linking 中处理多线程问题。

7.13.2.1 多线程环境中的 Reference Counting

如果在不同的线程之间拷贝 smart pointer, 就会造成“在无法预料时刻”“从不同的线程中”递增引用计数。

正如附录所说,“递增”并非“不可切割之操作”(所谓原子操作, atomic operation)。要想在多线程环境下递增或递减一个整数值,你必须使用 `ThreadingModel<T>::IntType` 型别,以及 `AtomicIncrement()` 和 `AtomicDecrement()` 函数。

这里事情变得有些棘手。更恰当地说是,如果想将线程和引用计数策略分离,事情变得棘手。

设计 policy-based class 时,我们需要将一个 class 分解为数个基本行为元素,并将每一个元素限制为一个独立的 template 参数。理想情况下 `SmartPtr` 会指定一个 Ownership policy 和一个 ThreadingModel policy, 并运用它们达成一份正确实作。

但是在“多线程引用计数”的情况下,事物被过于紧密地联系在一起。例如计数器的型别必须是 `ThreadingModel<T>::IntType`。这样一来你就不能使用 `operator++` 和 `operator--`, 必须使用 `AtomicIncrement()` 和 `AtomicDecrement()`。“线程”和“引用计数”融合在一起,很难将它们分开来。

最好的做法是将多线程结合到 Ownership policy 中。这样一来你就会有两份实作: `RefCounting` 和 `MultiThreadedRefCounting`。

7.13.2.2 多线程环境中的 Reference Linking

请看看 reference linking smart pointer 的析构函数。大致像下面这样:

```
template <class T>
class SmartPtr
{
public:
    ~SmartPtr() {
        if (prev_ == next_) {
            delete pointee_;
        }
        else {
            prev_->next_ = next_;
            next_->prev_ = prev_;
        }
    }
    ...
private:
    T* pointee_;
    SmartPtr* prev_;
```

```
SmartPtr* next_;  
};
```

在这个析构函数中，我们执行的是典型的 doubly linked list 删除动作。为了让实作更简单更快速，list 为环型结构，最后一个节点指向第一个节点。这样一来，对于任何 smart pointer 我们都不必测试 prev_ 和 next_ 是否为零。在只有一个元素的环型 list 中，prev_ 和 next_ 都是 this。

对于相互链接在一起的 smart pointers，如果有多个线程摧毁它，其析构函数必须具备不可切割性（原子性，atomic，亦即不得被其他线程中断）。否则另一个线程有可能中断这个 SmartPtr 的析构函数——也许就在更新 prev_>next 和更新 next_>prev 之间，于是造成那个线程在“被破坏了”的 list 上执行操作。

类似的分析适用于 SmartPtr 的 copy 构造函数和 assignment 操作符。这些函数必须具备不可切割性（原子性，atomic），因为它们要操纵“ownership list”。

有趣的是，在这里，我们无法应用 object-level 锁定语义。在附录中，锁定策略被划分为 class-level 和 object-level。class-level 锁定动作会对操作期间某个 class 的所有对象执行锁定。object-level 锁定动作只锁定和操作有关的那个对象。前一技术占用的内存少些（每个 class 只需一个 mutex），但会导致效率上的瓶颈。后者较沉重（每个对象需要一个 mutex），但速度较快。

我们无法对 smart pointers 实施 object-level 锁定，因为一个操作要处理多达三个对象：当前正在添加或删除的对象、ownership list 中的前一个对象和后一个对象。如果想引入 object-level 锁定，首先要考虑的是：每个 pointee 对象都必须有一个 mutex——因为每个 pointee 对象都有一个相应的 list。我们可以为每个对象动态分配 mutex，但这抹杀了 reference linking 相对于 reference counting 的主要优点。前者可爱之处正在于它不使用自由空间（free store）。

另外，我们还可以使用“侵入式”方案：pointee 对象保存着 mutex，smart pointer 操纵这个 mutex。但另一个合理而有效的选择——reference counted smart pointers——会让你否定这一方案。

总而言之，smart pointers 如果采用 reference counting 或 reference linking 技术，便会受多线程的影响。“多线程环境下安全（thread safe）”的 reference counting，需要整数原子操作；“多线程环境下安全”的 reference linking 则需要 mutexes。SmartPtr 只提供前者。

7.14 将一切组装起来

再没什么好说的了！现在是最快乐的时刻。到目前为止，我们已经分别讨论了每一个主题。现在我们要将所有的决策整理在一起，构成一个完整的 SmartPtr 实作品。

我将使用第 1 章介绍过的策略（strategy）：policy-based class design。每一个未有独立解决方案的设计要素，都会被转移到一个 policy 中。SmartPtr class template 接受所有这些 policies，作为一个个独立的 template 参数。SmartPtr 继承所有这些 template 参数，并允许相应的 policies 存储状态（state）。

让我们回顾前面各节内容，列出 SmartPtr 的变动点（variation points）。每一个变动点被转化

为一个 policy。

- **Storage policy** (7.3 节)。缺省情况下 stored type 是 T^* (T 是 `SmartPtr` 的第一个 template 参数), pointer type 也是 T^* , reference type 是 $T\&$ 。摧毁 pointee 对象的方式是使用 `delete` 操作符。
- **Ownership policy** (7.5 节)。较普遍的实作有 deep copy (深层拷贝)、reference counting (引用计数)、reference linking (引用链接) 和 destructive copy (摧毁式拷贝)。请注意, Ownership 和析构机制本身没有关系, 那是 Storage 的工作。Ownership 控制的是析构时刻 (moment)。
- **Conversion policy** (7.7 节)。某些应用程序需要向“底层原始指针型别”作自动转换; 其他应用则不需要。
- **Checking policy** (7.10 节)。这个 policy 控制 `SmartPtr` 的初始值是否合法, 以及对 `SmartPtr` 的提领是否合法。

其他问题并不值得为之提供专门的 policies 或做出优化方案:

- **address-of** (取址) 操作符 (第 7.6 节) 最好不要重载。
- 相等性测试和不等性测试应采用 7.8 节演示的技巧。
- 7.9 节的 ordering comparisons (次序比较) 操作符并未实作, 然而 Loki 特化了 `std::less` 以用于 `SmartPtr` 对象。用户可以定义 `operator<`, Loki 利用 `operator<` 来定义其他所有 ordering comparisons (次序比较) 操作符。
- Loki 为 `SmartPtr` 对象、pointee 对象或二者定义了具有正确常数性 (constness) 的实作。
- 对 array 并无特别支持, 但是有一个经过包装的 Storage 实作品可通过 `operator delete[]` 处理 array。

在以 smart pointers 为中心提出这些设计问题之后, 问题本身显得更易于理解和处理, 因为每一个问题都是分开讨论的。如果我们的实作能够分解这些问题并分别对其进行处理, 而不是一次性地挑战所有复杂性, 将会非常有益。

分而治之! 即使是在今天, 将 Julius Caesar 创造的这条古老原则运用于 smart pointers 身上, 我们也会受益无穷 (我敢打赌, 他当初绝对没有预计到这一点)。我们将问题分解为小型组件式类 (small component classes), 即所谓 policies (策略)。每个 policy class 只处理一个问题。`SmartPtr` 继承了所有这些 classes, 因而继承了它们所有的特性。这的确简单, 但正如你很快看到的那样, 它还具有惊人的灵活性。每一个 policy 也是一个 template 参数, 这意味着你可以搭配使用现有的 policy classes, 也可以构造你自己的版本。

`SmartPtr` 的第一个 template 参数是“被指对象”的型别, 接下去是一个个 policy。下面是 `SmartPtr` 的完整声明:

```
template
<
    typename T,
    template <class> class OwnershipPolicy = RefCounted,
    class ConversionPolicy = DisallowConversion,
```

```

template <class> class CheckingPolicy = AssertCheck,
template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;

```

在此声明中，policies 出现的次序是：你最常定制的 policies 被放在最前面。

以下四小节分别就四个 policies 的需求 (requirements) 进行探讨。所有 policies 都遵循一条原则：需具备“value 语义”，也就是说必须定义适当的 copy 构造函数和 assignment 操作符。

7.14.1 Storage Policy

Storage policy 提炼了 smart pointer 的结构，它提供型别定义，并存储实际的 pointee_ 对象。

如果 StorageImpl 是 Storage policy 的一份实作，storageImpl 是型别 StorageImpl<T> 的一个对象，那么表 7.1 所列的构件是合法的。下面是缺省之 Storage policy 的实作：

```

template <class T>
class DefaultSPStorage
{
protected:
    typedef T* StoredType;           // the type of the pointee_ object
    typedef T* PointerType;          // type returned by operator->
    typedef T& ReferenceType;        // type returned by operator*
public:
    DefaultSPStorage() : pointee_(Default()) {}
    DefaultSPStorage(const StoredType& p) : pointee_(p) {}
    PointerType operator->() const { return pointee_; }
    ReferenceType operator*() const { return *pointee_; }
    friend inline PointerType GetImpl(const DefaultSPStorage& sp)
    { return sp.pointee_; }
    friend inline const StoredType& GetImplRef(
        const DefaultSPStorage& sp)
    { return sp.pointee_; }
    friend inline StoredType& GetImplRef(DefaultSPStorage& sp)
    { return sp.pointee_; }
protected:
    void Destroy()
    { delete pointee_; }
    static StoredType Default()
    { return 0; }
private:
    StoredType pointee_;
};

```

除了上述的 DefaultSPStorage，Loki 还提供以下定义：

- **ArrayStorage**，它在 Release 中使用了 operator delete[]
- **LockedStorage**，它运用分层 (layering) 技术，为 smart pointer 提供“提领时锁定数据”的功能（见第 7.13.1 节）
- **HeapStorage**，它采用显式析构调用，并通过 std::free 释放数据

表 7.1 Storage policy 的合法构件

表达式 (Expression)	语义 (Semantics)
<code>StorageImpl<T>::StoredType</code>	实作品中实际存储的型别。缺省为 <code>T*</code>
<code>StorageImpl<T>::PointerType</code>	实作品中定义的指针型别。这是 <code>SmartPtr</code> 的 <code>operator-></code> 的返回型别。缺省为 <code>T*</code> 。使用 smart pointer 分层技术时可与 <code>StorageImpl<T>::StoredType</code> 不同 (见 7.3 和 7.13.1 节)
<code>StorageImpl<T>::ReferenceType</code>	引用型别。它是 <code>SmartPtr</code> 的 <code>operator*</code> 的返回型别。缺省为 <code>T&</code>
<code>GetImpl(storageImpl)</code>	传回 <code>StorageImpl<T>::StoredType</code> 对象
<code>GetImplRef(storageImpl)</code>	传回 <code>StorageImpl<T>::StoredType&</code> 对象, 如果 <code>storageImpl</code> 是 <code>const</code> , 传回之对象也是 <code>const</code>
<code>storageImpl.operator->()</code>	传回 <code>StorageImpl<T>::PointerType</code> 对象, 用于 <code>SmartPtr</code> 自身的 <code>operator-></code>
<code>storageImpl.operator*()</code>	传回 <code>StorageImpl<T>::ReferenceType</code> 对象, 用于 <code>SmartPtr</code> 自身的 <code>operator*</code>
<code>StorageImpl<T>::StoredType p;</code> <code>p = storageImpl.Default();</code>	传回缺省值 (通常为零)
<code>storageImpl.Destroy()</code>	摧毁 pointee 对象

7.14.2 Ownership Policy

Ownership policy 必须能够支持“侵入式引用计数”和“非侵入式引用计数”。因此, 就像 Keonig (1996) 的做法一样, Loki 用的是“显式函数调用”, 而非“构造函数/析构函数”技术。这样做的原因是, 成员函数可在任何时候被调用, 而构造函数和析构函数只能在特定时刻自动被调用。

Ownership policy 的实作版本接受一个 template 参数, 此即相应之指针型别 (pointer type)。SmartPtr 会将 `StoragePolicy<T>::PointerType` 传递给 `OwnershipPolicy`。请注意, `OwnershipPolicy` 的 template 参数是一个指针型别, 不是一个对象型别。

如果 `OwnershipImpl` 是 `Ownership` 的一份实作品, 而且 `ownershipImpl` 是 `OwnershipImpl<P>` 的一个对象, 那么表 7.2 所列构件是合法的。

表 7.2 Ownership policy 的合法构件

表达式 (Expression)	语义 (Semantics)
<pre>P val1; P val2 = OwnershipImpl. Clone(val1);</pre>	复制 (克隆) 一个对象。如果 OwnershipImpl 使用 “摧毁式拷贝”，这个动作会修改源值 (source value)
<pre>const P val1; P val2 = ownershipImpl. Clone(val1);</pre>	复制 (克隆) 一个对象
<pre>P val; ownershipImpl Release(val);</pre>	释放对象拥有权
<pre>P val; bool unique = ownershipImpl. IsUnique(val);</pre>	测试是否 value 被唯一引用 (uniquely referred)。如果 InUnique() 传回 true, SmartPtr 的析构函数会触发 Storage policy 的 Destroy() 成员函数
<pre>bool dc = OwnershipImpl<P> ::destructiveCopy;</pre>	表明 OwnershipImpl 是否使用摧毁式拷贝。如果是, SmartPtr 将使用 std::auto_ptr 中所使用的 “Colvin/Gibbons 技巧” (Meyers 1999)

下面是一份支持 reference counting 的 Ownership 实作版本:

```
template <class P>
class RefCounted
{
    unsigned int* pCount_;
protected:
    RefCounted() : pCount_(new unsigned int(1)) {}
    bool IsUnique() const
    {
        return *pCount_ == 1;
    }
    P Clone(const P& val)
    {
        ++*pCount_;
        return val;
    }
    bool Release(const P&)
    {
        if (--*pCount_) delete pCount_;
    }
    enum { destructiveCopy = false }; // see below
};
```

实作一份 reference counting 是十分容易的事。让我们来为 COM 对象提供一个 Ownership policy 实作。COM 对象有两个函数: `AddRef()` 和 `Release()`。进行最后一次 `Release()` 调用时, 对象将摧毁自身。你只需将 `Clone()` 导至 COM `AddRef()`, 将 `Release()` 导至 COM `Release()` 即可:

```
template <class P>
class COMRefCounted
{
public:
    static P Clone(const P& val)
    {
        val->AddRef();
        return val;
    }
    static bool Release(const P& val)
    {
        val->Release();
        return false;
    }
    enum { destructiveCopy = false }; // see below
};
```

Loki 定义了以下这么多的 Ownership 实作品:

- **DeepCopy**, 7.5.1 节有所介绍。DeepCopy 要求 pointee class 实作一个 `Clone()` 成员函数。
- **RefCounted**, 7.5.3 节和本节有所介绍。
- **RefCountedMT**, 这是 **RefCounted** 的多线程版本。
- **COMRefCounted**, 这是“侵入式引用计数”的变体, 本节作了介绍。
- **RefLinked**, 7.5.4 节有所介绍。
- **DestructiveCopy**, 7.5.5 节有所介绍。
- **NoCopy**, 未曾定义 `Clone()`, 因而禁止任何形式的拷贝动作。

7.14.3 Conversion Policy

Conversion policy 很简单: 它定义一个编译期 bool 常量, 用以表明 **SmartPtr** 是否允许“隐式转换至底层的指针型别”。

如果 **ConversionImpl** 是 **Conversion** 的一份实作品, 那么表 7.3 所列构件是合法的。

SmartPtr 的底层指针型别由其 **Storage policy** 决定, 亦即 `StorageImpl<T>::PointerType`。

正如你所想的那样, Loki 的确定义了两份 **Conversion** 实作品:

- **AllowConversion**
- **DisallowConversion**

表 7.3 Conversion Policy 的合法构件

表达式 (Expression)	语义 (Semantics)
<code>bool allowConv = ConversionImpl<P>::allow;</code>	如果 <code>allow</code> 为 <code>true</code> , <code>SmartPtr</code> 允许隐式转换至底层指针 型别

7.14.4 Checking Policy

正如 7.10 节所述, 为保持一致性, 检测 `SmartPtr` 对象的地点主要有两个: 初始化期间和提领 (`dereference`) 之前。检测过程中可能会使用 `assert`、异常、缓式初始化, 或干脆什么也不做。

`Checking policy` 操作于 `Storage policy` 的 `StoredType` 之上, 而不在 `PointerType` 之上。关于 `Storage` 的定义, 请参阅 7.14.1 节。

如果 `s` 是 `Storage policy` 实作品中定义的存储型别、`CheckingImpl` 是 `Checking` 的一份实作, 而且 `checkingImpl` 是个 `CheckingImpl<S>` 对象, 那么表 7.4 所列构件是合法的。

表 7.4 Checking Policy 的合法构件

表达式 (Expression)	语义 (Semantics)
<code>S value; checkingImpl.OnDefault(value);</code>	<code>SmartPtr</code> 总是在 <code>default</code> 构造函数中调用 <code>OnDefault()</code> 。 因此, 如果 <code>CheckingImpl</code> 未曾定义此函数, <code>default</code> 构造函数就会在编译期被禁绝
<code>S value; checkingImpl.OnInit(value);</code>	<code>SmartPtr</code> 会在调用构造函数时调用 <code>OnInit()</code>
<code>S value; checkingImpl.OnDereference (value);</code>	在 <code>operator-></code> 和 <code>operator*</code> 返回之前, <code>SmartPtr</code> 会 调用 <code>OnDereference()</code>
<code>const S value; checkingImpl.OnDereference (value);</code>	在 <code>const</code> 版的 <code>operator-></code> 和 <code>operator*</code> 返回之前, <code>SmartPtr</code> 会调用 <code>OnDereference()</code>

Loki 定义了下列 `Checking` 实作品:

- `AssertCheck`, 通过 `assert` 在提领之前对 `value` 进行检测。
- `AssertCheckStrict`, 通过 `assert` 在初始化时对 `value` 进行检测。
- `RejectNullStatic`, 未曾定义 `OnDefault()`。所以只要客端用到 `SmartPtr` 的 `default` 构造函数, 都会产生编译期错误。

- `RejectNull`，如果提领 `null` 指针，会抛出一个异常。
- `RejectNullStrict`，不允许 `null` 指针作为初始值（如是，亦会抛出一个异常）。
- `NoCheck`，继承 C 和 C++ 的广泛传统——根本不做检测。

7.15 摘要

恭喜，你读完了本书篇幅最长、内容最繁杂的一章，但愿努力有所回报。现在你应该已经理解 `smart pointers` 的很多东西，并得到了一个功能全面、配置灵活的 `SmartPtr class template`。

`Smart pointers` 在语法和语义上模拟了 `built-in pointers`。此外，它还完成后者无法完成的大量工作。这些工作包括“拥有权的管理”和“无效值的检测”。

`Smart pointers` 这一概念超脱真实指针所具有的行为：它可以广义化（`generalized`）为所谓 `smart resources`，例如 `monikers`（一种不具有指针语法的 `handle`，在允许资源访问方面采用类似指针的做法）。

对于难以手工管理的事情，`smart pointers` 可以出色地自动处理；因而，在设计一个成功、坚固的应用程序时，`smart pointers` 是必不可少的辅助工具。它们很小，但它们可以影响到一个项目的成败——或者更常见的是，它会影响程序是“正确的”还是“像漏斗一样地四处泄漏资源”。

这就是为什么 `smart pointer` 的实作者必须尽可能投入大量注意力和精力的原因。这种投入会得到长期的回报。`Smart pointers` 的用户应该了解他所用的 `smart pointers` 的规则，并且在使用过程中遵守这些规则。

在我们提供的 `smart pointers` 实作品中，设计重心是将功能分解为独立的 `policies`（策略），主要的 `class template SmartPtr` 可搭配使用这些 `policies`。这种设计的确是可行的，因为每一个 `policy` 都实作出一个定义明确的接口。

7.16 SmartPtr 要点概览

- `SmartPtr` 声明如下：

```
template
<
    typename T,
    template <class> class OwnershipPolicy = RefCounted,
    class ConversionPolicy = DisallowConversion,
    template <class> class CheckingPolicy = AssertCheck,
    template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

- `T` 是 `SmartPtr` 所指型别，可以是基本型别或用户自定义型别，也可以是 `void` 型别。
- 对于其余的 `class template` 参数（`OwnershipPolicy`、`ConversionPolicy`、`CheckingPolicy` 和 `StoragePolicy`），你可以实作出自己的 `policies`，也可以选择 7.14.1~7.14.4 节提到的缺

省实作品。

- **OwnershipPolicy** 负责控制“拥有权管理策略”。你可以选择预先定义好的 **DeepCopy**、**RefCounted**、**RefCountedMT**、**COMRefCounted**、**RefLinked**、**DestructiveCopy**、**NoCopy** classes 中的一个。7.14.2 节对此作了介绍。
- **ConversionPolicy** 负责控制“是否允许隐式转换至 pointee 型别”。缺省是禁止隐式转换。无论是否允许转换，你还是可以调用 `GetImpl()` 来取用 pointee 对象。你可以选择使用 **AllowConversion** 或 **DisallowConversion** 实作品（7.14.3 节）。
- **CheckingPolicy** 定义“出错检查策略”。缺省实作品有 **AssertCheck**、**AssertCheckStrict**、**RejectNullStatic**、**RejectNull**、**RejectNullStrict**、**NoCheck**（7.14.4 节）。
- **StoragePolicy** 就“如何存储和访问 pointee 对象”进行详细定义。缺省为 **DefaultSPStorage**，当通过型别 T 具现化时，它所定义的 reference type 是 $T\&$ ，stored type 是 T^* ，`operator->` 的返回型别也是 T^* 。Loki 定义的其他 storage type 还有 **ArrayStorage**、**LockedStorage**、**HeapStorage**（7.14.1 节）。

8

Object Factories

对象工厂

面向对象程序（object-oriented program）通过继承（inheritance）和虚函数（virtual functions）获得功能强大的抽象性（abstractions）和良好的模块性（modularity）。由于我们得以延至执行期才决定“哪个具体函数将被调用”，因此多态（polymorphism）得以提高二进制码的复用性和扩充性。执行期系统自动将虚成员函数分派（dispatches）给合适的派生对象，让你得以利用基本而原始的多态来完成复杂行为。

你可以在任何一本讲授面向对象技术的书籍中找到类似上面的文字描述。这里之所以老调重弹，是因为我想就 *steady mode*（稳定模式）中的美好状况和 *initialization mode*（初始模式）中令人讨厌的情形作个对比；后一状况下你必须以多态方式产生对象。

在 *steady mode* 之下你已经拥有了指向多态对象的 *pointer* 或 *reference*，你可以对着它们调用成员函数，其动态型别完全确知（虽然调用者可能并不清楚）。某些情况下，对象的产生需要同样的灵活性，也就是说我们需要所谓的虚构造函数（virtual constructors）这种矛盾东西。是的，如果“待产物”的信息是动态的，无法直接被 C++ 构造函数所用，你就需要虚构造函数。

通常，多态对象是经由 *new* 操作符在自由空间（free store）中产生的：

```
class Base { ... };
class Derived : public Base { ... };
class AnotherDerived : public Base { ... };
...
// Create a Derived object and assign it to a pointer to Base
Base* pB = new Derived;
```

这里的问题在于，调用 *new* 操作符时出现了 *Derived* 这一确凿的型别名称。某种程度上，这里的 *Derived* 很像我们向来不提倡的魔术常量。如果想产生 *AnotherDerived* 对象，你必须来到以上语句，以 *AnotherDerived* 替换 *Derived*。你无法让上述 *new* 操作符以更“多态性”的方式工作：你必须传给它一个型别，而该型别在编译期必须完全确知。

这表现出 C++ 之中“对象生成”和“虚函数调用”的一个根本不同。虚函数是流动的、动态的——你可以在不改变调用端 (call site) 的情况下改变它们的行为。相反的，每一个对象的生成都是一段笨拙、静态绑定的死板代码。影响所及，“虚函数调用”仅仅将调用者绑定于接口 (亦即 base class) 之上。面向对象技术总是企图打破“对具象型别 (concrete type) 的依存性”，然而，至少在 C++ 中，“对象的生成”却将调用者绑死于最底层具象派生类。

其实从概念上来说，事情之所以会这样是有很多道理的：即使在日常生活中，产生某物和处理某物也大不相同。因而当你着手生成一个对象时，你理当确切知道你干什么，但有时候：

- 你想将这种确切信息留给另一物体 (entity)。例如你可能不直接调用 new，而是调用某一具有更高级别的对象中的虚函数 Create，从而让客户可以通过多态性来改变行为。
- 你的确了解型别相关信息，但这种信息无法以 C++ 表达。例如你可能拥有一个字符串，其中包含 "Derived"；因此你的确知道必须产生一个型别为 Derived 的对象，但你却无法将一个“内含型别名称”的字符串 (而非型别名称本身) 传给 new。

这两点正是 object factory (对象工厂) 所要解决的基本问题。本章将对此进行详细讨论。本章包括以下主题：

- 藉由范例，说明哪些情况下需要使用 object factory。
- 为什么虚构造函数 (virtual constructors) 在 C++ 中天生难以实现。
- 如何通过 *substituting values for types* (以实值代替型别) 的方式产生对象。
- 泛型 object factory 的实作。

本章结束前，我们将设计出一个泛化的 object factory。藉由产品类型、生成方法、产品标识法等等，你可以对它进行大范围的定制 (customized)。你可以利用本书介绍的其他组件来产生 (组装) 这个“工厂”，例如运用第 6 章的 Singleton 获得一个 (对你的应用程序而言) 适用的 object factory，或运用第 5 章的 Functor 调整工厂行为。我还将介绍一个克隆 (clone) 工厂，可对任意型别之对象进行克隆 (翻制)。

8.1 为什么需要 Object Factories

在两种基本情况下我们需要 object factory。第一种情况是，程序库不仅需要“操纵”用户自定义对象，还需要“产生”它们。举个例子，想象你要开发一个“多窗口文档编辑器”框架 (framework)。为了使它易于扩充，你提供一个抽象类 Document，用户可以由它派生出其他 classes，例如 TextDocument 或 HTMLDocument。这个框架的另一个组件可能是 DocumentManager class，它维护“所有已开启文档”的清单。

这里要引入一条好规则：DocumentManager 应该知道存在于程序中的每一份文档。因此，只要产生新文档，就必须被添加到 DocumentManager 文档清单中，两个操作紧密联系。当两个操作如此紧密联系时，我们最好将它们放到同一个函数中，不要分开执行它们：


```
class DocumentManager
{
    ...
public:
    Document* NewDocument();
private:
    virtual Document* CreateDocument() = 0;
    std::list<Document*> listOfDocs_;
};

Document* DocumentManager::NewDocument()
{
    Document* pDoc = CreateDocument();
    listOfDocs_.push_back(pDoc);
    ...
    return pDoc;
}
```

成员函数 `CreateDocument()` 取代了对 `new` 的调用。`NewDocument()` 不能使用 `new` 操作符，因为撰写 `DocumentManager` 时并不知道将来要产生什么具体文档。为了使用这个框架，程序员必须继承 `DocumentManager` 并改写其中的虚（很可能是纯虚）函数 `CreateDocument()`。GoF 著作（Gamma 等, 1995）将 `CreateDocument` 称为 *factory method*（工厂方法）。

由于 *derived class* 知道即将产生之文档的确切型别，因而可以直接调用 `new` 操作符。如果采用这种方法，框架本身便无需知道型别信息，只需和 *base class* `Document` 打交道就好。改写动作非常简单，基本上只需包含一个 `new` 调用，像这样：

```
Document* GraphicDocumentManager::CreateDocument()
{
    return new GraphicDocument;
}
```

通过这个框架构造出的应用程序可能需要支持多种文档（例如 `bitmap` 图形和 `vector` 图形）。这种情况下，被改写的 `CreateDocument()` 函数可以向用户显示一个对话框，询问将欲产生的文档的具体类型。

上述框架如欲开启前次保存于磁盘的文档，会带来“object factory 必须存在”的第二个（同时也更复杂）的情况。当你将一个对象保存于文件时，你必须以字符串、整数或某种标识符保存其实际型别。这种情况下型别相关信息虽然存在，但其存在形式不允许你直接产生 C++ 对象。

这种情形反映出的一般概念是，“待产物”的型别相关信息被推迟至执行期：由终端用户输入，或从持久存储器（*persistent storage*）或网络连接等读取。与多态相比，此处“将型别绑定于实值（*binding of types to values*）”的动作被推得更远。使用多态时，操纵对象者并不知道对象的确切型别，但对象本身具有明确型别。然而当程序从某种存储介质读取对象时，型别于执行期“孤身”而至。你必须将型别相关信息转换为对象。最终你还必须通过一个虚函数从存储介质

中读取对象——一旦已经产生了一个空对象，这是轻而易举的事。

从单纯的型别信息到实质对象的生成，最终再将“动态的”信息改装为“静态的”C++ 型别，这是建立 object factory (对象工厂) 的一个重要议题。下一节我们将集中讨论这个议题。

8.2 Object Factories in C++: Classes 和 Objects

拿出解决方案之前，我们需要先很好地理解问题。本节试图回答以下问题：C++ 构造函数为什么这么死板？为什么没有办法藉由语言本身灵活地产生对象？

有趣的是，在寻找这个问题的答案时，我们会直接面对 C++ 型别系统的一个重要决策。为了弄清楚以下语句为什么如此死板：

```
Base* pB = new Derived;
```

我们必须回答两个相关问题：什么是 class？什么是 object？这是因为上述语句的症结在于 Derived——它是一个 class 名称，但我们却希望它产生一个实值 (value)，也就是一个 object。

在 C++ 中 classes 和 objects 是不同的东西。classes 由程序员产生，objects 由程序产生。你无法在执行期产生新的 classes，你也无法在编译期产生 objects。classes 并不具有第一级状态 (first-class status)：你无法拷贝一个 class，将它保存于变量中，或是从某个函数中传回。

但是在某些语言中，classes 就是 objects。在那些语言中，具有某些属性的 objects 习惯上会被视为 classes。因此，在那些语言中，你可以于执行期产生新的 classes，拷贝一个 class，将它保存于变量，等等。如果 C++ 也是这样一种语言，你就可以写出下面这样的代码：

```
// Warning--this is NOT C++
// Assumes Class is a class that's also an object
Class Read(const char* fileName);
Document* DocumentManager::OpenDocument(const char* fileName)
{
    Class theClass = Read(fileName);
    Document* pDoc = new theClass;
    ...
}
```

也就是说，我们可以将一个型别为 Class 的变量传递给 new 操作符。在这种方式下，“将一个已知的 class 名称传递给 new”和“使用一个写死的 (hardcoded) 常数”是一样的。

这种动态语言为了灵活性而损失了某些型别安全性和效率，因为静态型别 (static typing) 是优化的重要源泉。C++ 采取相反的做法，它坚持静态型别系统，但它还是希望在前述框架中尽可能提供最大灵活性。

总而言之，在 C++ 中，object factories 的建立是一个很复杂的问题。C++ 中的 type 和 value 之间存在着裂缝：value 拥有 type 所表示的属性，但 type 无法靠自身存活。如果想通过完全动态的方式来产生 object，你就得有某种方法来表达和传递纯粹的 type，并能够根据请求，从 type 构造出 value。这一点无法做到，所以你必须藉由某种方式将 type 表示为 object ——整数或字

字符串，等等，然后采用某种技巧拿 value 换取正确的 type，最后再运用那个 type 产生 object。在静态类型语言中，这种“object-type-object”交易行为对 object factories 而言是一种基石。

我们将“用以标识型别”的对象称为“型别标识符 (type identifier)”（请不要和 `std::typeid` 混淆了）。型别标识符可以帮助 object factory 产生出具有合适型别的对象。正如后文所演示，有时候你需要在“型别标识符”和“对象”之间进行交换，但你不确知你拥有什么对象，或是将得到什么对象。这很像一则童话故事：你不确切知道令牌 (token) 如何运作（试图理解它有时候会很危险），但你将它交给巫师，巫师会交换给你一个值钱的宝贝。至于魔法如何运作，其细节必须由巫师（也就是这儿所谓的 **factory**，工厂）封装起来。

以下我将对一个可解决具体问题的简单工厂进行分析，尝试其各种实作，然后从中提炼泛型性，构筑出一个 **class template**。

8.3 实现一个 Object Factory

假设你正在编写一个简单的绘图程序，该程序允许使用者编辑简单的向量图形（包括线、圆、多边形等等）³⁰。如果采用典型的面向对象风格，你可以定义一个 **abstract class Shape**，让所有图形都派生于它：

```
class Shape
{
public:
    virtual void Draw() const = 0;
    virtual void Rotate(double angle) = 0;
    virtual void Zoom(double zoomFactor) = 0;
    ...
};
```

然后你可能会定义一个包含复杂图形的 **Drawing class**，其中实质保存着一个集合 (collection)（例如 **list**、**vector**，或一个阶层体系结构），其内所含都是 **pointer to Shape**，并提供某些操作，将复杂图形视为一个整体来处理。两个典型的操作是：(1) 将图形保存为文件，(2) 从先前保存的文件中取出图形。

图形的保存很简单：只需提供一个纯虚函数，例如 `Shape::Save(std::ostream&)`。然后 `Drawing::Save` 看起来会像这样：

```
class Drawing
{
public:
```

³⁰ 这种 “Hello, world” 式的设计是面试 C++ 人员的一道好题目。很多面试官或许可以构思出这样一个设计，却很少有人知道如何实现文件的装载 (loading)，而那是非常重要的一项作业。

```

void Save(std::ofstream& outFile);
void Load(std::ifstream& inFile);
...
};

void Drawing::Save(std::ofstream& outFile)
{
    write drawing header
    for (each element in the drawing)
    {
        (current element) -> Save(outFile);
    }
}

```

这个 Shape-Drawing 示例常常出现在 C++ 书籍中, 包括 Bjarne Stroustrup 的经典著作 (Stroustrup 1997)。但是, 当进行到从文件装载图形时, 大多数 C++ 入门书籍都会停止脚步, 原因是“具有独立的图形对象”的良好模型被破坏了。若要讲述对象读取的繁杂细节, 需要一大段插曲, 因而往往理所当然地被回避了。但这正是我们想实现的功能, 所以我们必须面对挑战。一个直截了当的做法是, 要求每一个从 Shape 派生的对象都在文件起始处保存一个整数标识符。每个对象都有自己独一无二的 ID。那么, 文件的读取将会像这样:

```

// a unique ID for each drawing object type
namespace DrawingType
{
    const int
        LINE = 1,
        POLYGON = 2,
        CIRCLE = 3
};

void Drawing::Load(std::ifstream& inFile)
{
    // error handling omitted for simplicity
    while (inFile) {
        // read object type
        int drawingType;
        inFile >> drawingType;

        // create a new empty object
        Shape* pCurrentObject;
        switch (drawingType) {
            using namespace DrawingType;
        case LINE:
            pCurrentObject = new Line;
            break;
        case POLYGON:
            pCurrentObject = new Polygon;
            break;

```

```

    case CIRCLE:
        pCurrentObject = new Circle;
        break;
    default:
        handle error--unknown object type
}
// read the object's contents by invoking a virtual fn
pCurrentObject->Read(inFile);
add the object to the container
}
}

```

这的确是一个 object factory。它从文件中读取一个型别标识符，根据该符号产生相应的对象，并调用虚函数将该对象的内容从文件中读取出来。唯一的问题是，它违反了面向对象的最重要规则：

- 它基于型别标记 (type tag) 执行了 switch 语句，因而带有 switch 语句的相应缺点，这正是面向对象程序竭力消除的东西。
- 它在一个源码文件中收集所有关于 Shape 派生类的相关信息；这同样也是我们应该竭力避免的。理由是，对于所有可能的图形，Drawing::Save 的实作文件都因此必须包含其头文件，这就造成了编译依存性和维护上的瓶颈。
- 它难以扩充。假设要在系统中增加一种新图形，例如 Ellipse。那么，除了要产生这个 class，还必须在命名空间 DrawingType 中增加一个独一无二的整数常量，并在保存 Ellipse 对象时写入这个常量；还必须在 Drawing::Save 中为 switch 语句添加一个比对标记。这真是个糟糕的选择，远远偏离了这个架构最初的承诺——classes 之间完全隔离。而这一切都仅仅是为了一个函数！

我们希望产生一个这样的 object factory：它能完成我们的工作，但不存在以上缺点。首先要达到的一个实际目标是，拿掉 switch 语句，使 Line 的生成语句可以放在 Line 的实作文件中——对 Polygon 或 Circle 情况也一样。

要想将代码片段集成起来并操控之，常用的一个手法是函数指针——就像第 5 章讨论的那样。此处“可定制的代码单元 (unit of customizable code)”（也就是 switch 语句中的每一个入口）可以抽取至某个带有如下形式的函数内：

```
Shape* CreateConcreteShape();
```

我们的“工厂”维护着一个由函数指针组成的集合 (collection)，所有函数指针所指向的函数都具有以上形式。此外，ID 和“用于产生相应对象”的函数指针之间必须对应。因此，我们需要一个关联式集合 (associative collection) ——map 是很好的选择。针对特定的型别标识符，map 可以提供相应的函数取用机会——这正是 switch 语句所提供的功能。此外，map 还具有可伸缩性 (scalability)，这是编译期便被固定住的 switch 语句无法提供的弹性。map 可于执行期增长，可动态添加元素（那将是 ID 和函数指针的组合），这些正是我们所需要的。我们可以

从一个空 `map` 开始，然后令每一个“Shape 派生对象”为 `map` 添加一个元素。

为什么不使用 `vector`？ID 是整数，所以我们应该可以维护一个 `vector`，让 ID 成为 `vector` 的索引。那的确更简单更快速，但这里使用 `map` 还是更好一些。因为 `map` 并不要求其索引是连续值，而且 `map` 更具一般性——`vectors` 只能使用整数索引，`maps` 却可以拿任何有序型别作为索引。当我们准备将我们的示例泛化时，这一点很重要。

现在可以开始设计 `ShapeFactory` class 了，这个 class 负责管理所有 Shape 派生对象的生成。

实作 `ShapeFactory` 时我将运用 C++ 标准程序库提供的 `map`，亦即 `std::map`：

```
class ShapeFactory
{
public:
    typedef Shape* (*CreateShapeCallback)();
private:
    typedef std::map<int, CreateShapeCallback> CallbackMap;
public:
    // Returns 'true' if registration was successful
    bool RegisterShape(int ShapeId, CreateShapeCallback CreateFn);
    // Returns 'true' if the ShapeId was registered before
    bool UnregisterShape(int ShapeId);
    Shape* CreateShape(int ShapeId);
private:
    CallbackMap callbacks_;
};
```

这是一个“可伸缩工厂”（scalable factory）的基本设计。这个工厂具有可伸缩性，每次你向系统添加一个新的“Shape 派生类”时，不必修改它的代码。`ShapeFactory` 将职责划分清楚：每个新 Shape 都必须对工厂注册，也就是调用 `RegisterShape`，并将“整数标识”和“生成函数的指针”传递给它。通常，生成函数只有一行，大致像这样：

```
Shape* CreateLine() {
    return new Line;
}
```

`Line` 实作码必须将上述生成函数注册给程序所使用的 `ShapeFactory`——可见 `ShapeFactory` 往往是个可以在全局范围内被取用的对象³¹。注册动作一般是通过启动码（startup code）完成的。`Line` 和 `ShapeFactory` 的完整连接如下：

```
// Implementation module for class Line
// Create an anonymous namespace
// to make the function invisible from other modules
namespace
```

³¹ 这让我们想起 object factories 和 singletons 之间的联系。的确，多数情况下 factories 是 singletons。本章后续部分我们会讨论如何将第 6 章实现的 singletons 运用到 factories 中。

```

{
    Shape* CreateLine() {
        return new Line;
    }
    // The ID of class Line
    const int LINE = 1;
    // Assume TheShapeFactory is a singleton factory
    // (see Chapter 6)
    const bool registered =
        TheShapeFactory::Instance().RegisterShape(LINE, CreateLine);
}

```

有了 `std::map` 提供的便利性，实作 `ShapeFactory` 很容易。基本上 `ShapeFactory` 的成员函数只是将调用转发给 `callback_`：

```

bool ShapeFactory::RegisterShape(int shapeId,
    CreateShapeCallback createFn)
{
    return callbacks_.insert(
        CallbackMap::value_type(shapeId, createFn)).second;
}

bool ShapeFactory::UnregisterShape(int shapeId)
{
    return callbacks_.erase(shapeId) == 1;
}

```

如果你不熟悉 `std::map` class template，上述代码或许需要一些额外说明：

- `std::map` 中保存的是成对 (pairs) 的 keys 和 values。本例的 key 是图形的整数 ID，value 为函数指针。因此，pair 的型别应该是 `std::pair<const int, CreateShapeCallback>`。调用 `insert()` 时必须传递一个具有这样型别的对象。由于这写起来相当麻烦，所以最好使用 `std::map` 提供的 typedef，它为上述 pair 型别提供了一个方便的名称——`value_type`。你也可以使用 `std::make_pair`，这是另一种选择。
- 我们调用的 `insert()` 成员函数会传回另一个 pair，其中包含一个迭代器（指向刚安插进去的元素）和一个 bool；如果该元素的实值以前并不存在，那么 bool 为 true，否则为 false。调用 `insert()` 之后，`.second` 便取得了上述这个 bool，并直接传回，省得我们再产生一个带有名称的临时变量。
- `erase()` 会传回被删除的元素个数。

根据传入的 ID，`CreateShape()` 会找到对应的函数指针并调用之。如果出现错误，它会抛出异常。以下是其程序代码：

```

Shape* ShapeFactory::CreateShape(int shapeId)
{
    CallbackMap::const_iterator i = callbacks_.find(shapeId);

```

```
if (i == callbacks_.end()) {  
    // not found  
    throw std::runtime_error("Unknown Shape ID");  
}  
// Invoke the creation function  
return (i->second)();  
}
```

让我们看看这个简单的 class 给我们带来了什么。我们不再依赖庞大而仿佛无所不知的 switch 语句，相反地我们获得了一个动态体制，它要求每一种型别的对象都要对工厂进行注册。这就将职责从某个集中点转移到了它们所属的每一个 concrete class（具象类）身上。现在，如果想要定义新的 Shape 派生类，你可以只“增加”新文件，而不必“修改”旧文件。

8.4 型别标识符（Type Identifiers）

遗留下来的唯一问题就是对型别标识符的管理。同样地，型别标识符的增加也需要一定的规则和集中控管。只要增加一个新的图形类，你都必须检查每一个既有的型别标识符，然后加上一个不起冲突的标识符。如果存在冲突，那么同一个 ID 的第二次 RegisterShape() 会失败，你将无法产生该种对象。

为了解决这个问题，我们可以选择比 int 更具丰富含义的型别来表达标识符。这个型别不必是个整数，只要它能够成为 map 中的键值即可，也就是只要它支持 operator== 和 operator< 即可（这就是选择 map 而非 vector 带给我们的快乐）。例如我们可以将型别标识符保存为字符串，并建立一条规则：每一个 class 标识符都以 class 名称来表示：Line 的标识符是 "Line"，Polygon 的标识符是 "Polygon"，等等。这就将名称冲突的可能性降至最低，因为 class 的名称是唯一的。

如果你是一个连周末也喜欢研究 C++ 的人，上一段内容也许可以给你一些灵感。让我们使用 type_info 吧！std::type_info class 是 C++ 提供的执行期型别信息（RTTI）的一部分。对着一个型别或一个表达式调用 typeid 操作符，你便可以获得一个 reference to std::type_info。其中很棒的是 std::type_info 提供了一个成员函数 name()，可以传回型别名称（采用 const char* 型式），该名称对程序员来说是可读的。在你的编译器中你或许可以看到 typeid(Line).name() 指向 "class Line" 字符串。这正是我们想要的。

问题是这并不适用于所有 C++ 编译器。type_info::name() 的定义方式使它除了适合调试（例如在一个调试主控台中将它输出）之外，别无他用。我们无法保证这个字符串就是 class 的实际名称。更糟糕的是我们无法保证这个字符串在整个程序中独一无二（是的，通过 std::type_info::name()，你可能面对两个不同的 classes 却获得相同的名称）。更糟的是没有人能够保证那个型别名称历经时间之后仍然唯一。是的，我们无法保证 typeid(Line).name() 在不同的程序执行期间指向相同的字符串。持久性（persistence）的实现是 object factories 的一项重要应用，但 std::type_info::name() 不具持久性。因此，虽然 std::type_info 似乎对我们的 object factories 很有用，但它其实不是解答。

回到“型别标识符 (type identifier)”的管理上头。型别标识符的生成有一个分权 (decentralized) 解法, 也就是使用“可产生独一无二数值”的生成器——例如随机数生成器或随机字符串生成器。每当程序中增加一个新 class, 你都使用这个生成器, 然后将生成的随机值写死于源码文件中, 永远不改变它³²。这听起来好象很脆弱, 但请你想一想, 如果你有一个随机字符串生成器, 它在 1000 年内出现重复值的机率是 10^{-20} , 那么, 与一个使用了“完美”工厂的程序相比较, 你的出错率可能更小。

这里唯一可以得出的一个结论是, 型别标识符的管理不是 object factory 本身的工作。C++ 无法保证得到唯一而持久的型别 ID; 型别 ID 的管理是一个必须留待程序员解决的“语言之外” (extra-linguistic) 的难题。

我已经介绍了一个典型的 object factory 应具备的所有要素, 并且有了一份初步实作品。现在让我们进行下一步, 由具体进而抽象。经过进一步深刻理解之后, 我们会再由抽象回到具体。

8.5 泛化 (Generalization)

让我列举先前对 object factories 的讨论所涉及的要素。在组装 generic object factory (泛型对象工厂) 前, 这是必要的智力测验。

- 具体产品 (Concrete product)。工厂以对象的形式交付产品。
- 抽象产品 (Abstract product)。产品继承自某一基础型别 (前例中的 Shape)。一个产品是一个对象, 其型别隶属于某个继承体系。继承体系中的基础型别 (base type) 就是抽象产品。工厂以多态方式运作, 传回一个指向抽象产品的指针, 不带有具体产品的型别信息。
- 产品型别标识符 (Product type identifier)。此物用来标识具体产品的型别。如前所述, 由于 C++ 静态型别系统之故, 你必须先有一个型别标识符, 才能生成一个产品。
- 产品生产者 (Product creator)。函数或仿函数专门用来生成某一类对象。我们通过函数指针来模塑 (modeled) 产品生产者。

泛型工厂 (generic factory) 将协调这些元素, 提供定义良好的接口, 并针对经常被使用的情况提供某些缺省功能。

以上列举的每一个概念几乎都可以转化为 Factory template class 的 template 参数。只有一个例外: 工厂不必知道具体产品, 否则对新增的每一个具体产品我们就得有不同的 Factory 型别——而我们此刻正竭力让 Factory 与具体型别分离。所以, 只有面对不同的抽象产品, 我们才需要不同的 Factory 型别。

³² Microsoft's COM factory 就是使用这种做法。他们有一个算法用来为 COM 对象产生独一无二的 128-bit 标识符 (称为 globally unique identifiers, GUID)。这个算法依赖网路卡序号 (network card serial number) 的唯一性。如果没有网路卡, 则运用日期、时间以及其他高度变异的机器状态。

既然如此，我们可以写出以下代码；它们反映出迄今为止我们所掌握的设计思想：

```
template
<
    class AbstractProduct,
    typename IdentifierType,
    typename ProductCreator
>
class Factory
{
public:
    bool Register(const IdentifierType& id, ProductCreator creator)
    {
        return associations_.insert(AssocMap::value_type(id, creator)).second;
    }
    bool Unregister(const IdentifierType& id)
    {
        return associations_.erase(id) == 1;
    }
    AbstractProduct* CreateObject(const IdentifierType& id)
    {
        typename AssocMap::const_iterator i = associations_.find(id);
        if (i != associations_.end())
        {
            return (i->second)();
        }
        handle error
    }
private:
    typedef std::map<IdentifierType, AbstractProduct> AssocMap;
    AssocMap associations_;
};
```

剩下的只是错误处理了。如果没有找到一个“已注册”的生产者 (*creator*)，我们应该抛出异常吗？或者应该传回 `null` 指针？或是终止程序？或是动态装载某个程序库后立刻注册并再次执行该操作？实际措施取决于具体情况；在特定场合下这些行为各具意义。

我们的泛型工厂应该允许用户定制，使得以执行上面任何一种行为；此外，也应该提供一套合理的缺省行为。所以，“错误处理”应该从成员函数 `CreateObject()` 中抽取出来，放进一个独立的 `FactoryError` policy (第 1 章) 中。这个 policy 只定义一个函数：`OnUnknownType`；`Factory` 会给这个函数一个合适的机会（和足够的消息），让它作出合理选择。

`FactoryError` policy 的定义非常简单。它是一个 `template`，带有两个参数：`IdentifierType` 和 `AbstractProduct`。如果 `FactoryErrorImpl` 是 `FactoryError` 的一份实作品，那么以下表达式必然可用：

```
FactoryErrorImpl<IdentifierType, AbstractProduct> factoryErrorImpl;
IdentifierType id;
AbstractProduct* pProduct = factoryErrorImpl.OnUnknownType(id);
```

Factory 将 **FactoryErrorImpl** 视为万不得已的解决方案：如果 `CreateObject()` 无法在其内部的 `map` 中找到相应的东西，才会寻求使用 `FactoryErrorImpl<IdentifierType, AbstractProduct>::OnUnknownType`，以求获得指向抽象产品 (*abstract product*) 的指针；如果 `OnUnknownType()` 抛出异常，异常会从 **Factory** 中传播开来，否则 `CreateObject()` 就仅仅传回 `OnUnknownType()` 所传回的东西。

让我们把这些想法写成代码，作出相应的修改（以粗体字呈现）：

```
template
<
    class AbstractProduct,
    typename IdentifierType,
    typename ProductCreator,
    template<typename, class>
        class FactoryErrorPolicy
>
class Factory
    : public FactoryErrorPolicy<IdentifierType, AbstractProduct>
{
public:
    AbstractProduct* CreateObject(const IdentifierType& id) {
        typename AssocMap::const_iterator i = associations_.find(id);
        if (i != associations_.end()) {
            return (i->second)();
        }
        return OnUnknownType(id);
    }
private:
    ... rest of functions and data as above ...
};
```

FactoryError 的缺省实作码会抛出一个异常。代表该异常的那个 `class` 最好和其他所有型别截然不同，这么一来客端 (client) 就可以单独检测它，并给出相应对策。这个 `class` 还应该继承自某一标准的异常类 (exception class)，这么一来客端就可以藉由一个 `catch block` 捕获所有类型的错误。**DefaultFactoryError** 定义了一个继承自 `std::exception` 的嵌套 (nested, 又称 inner, 内隐) 异常类，名为 **Exception**³³。

³³ 没必要特别取一个与众不同的名称 (例如 **FactoryException**)，因为这个型别已经在 **DefaultFactoryError** `class template` 内部，外界不可得见。

```

template <class IdentifierType, class ProductType>
class DefaultFactoryError
{
public:
    class Exception : public std::exception
    {
    public:
        Exception(const IdentifierType& unknownId)
            : unknownId_(unknownId)
        {
        }
        virtual const char* what()
        {
            return "Unknown object type passed to Factory.";
        }
        const IdentifierType GetId()
        {
            return unknownId_;
        }
    };
private:
    IdentifierType unknownId_;
};

protected:
    StaticProductType* OnUnknownType(const IdentifierType& id)
    {
        throw Exception(id);
    }
};

```

更高级的 **FactoryError** 实作码可能会查找型别标识符，传回一个指向有效对象的指针，或是传回一个 **null** 指针（如果情况不适合使用异常），或抛出某个异常对象，或终止程序。你可以调整其行为——只要定义新的 **FactoryError** 实作码，并指定为 **Factory** 的第 4 引数即可。

8.6 细节琐务

事实上 **Loki** 的 **Factory** 实作品中并未使用 `std::map`。它使用一个“掺杂其他功能”的 **map** 替代品：**AssocVector**，此物针对“少量安插但频繁查找”的情况做了优化，而那正是 **Factory** 的典型运用型态。本书第 11 章详细讨论 **AssocVector**。

在 **Factory** 的最初设计中，**map** 型别是可供定制的，因为它是 **template** 参数。但 **AssocVector** 往往符合现实需要；此外，可以说，将标准容器当作 **template** 参数并不是一种标准手法，因为标准容器的实作者可以任意增加更多 **template** 引数——只要为它们都提供了缺省值。

现在，让我们把注意力集中在 **ProductCreator** 这个 **template** 参数上。它的主要条件是：需具备有效行为（接受不带引数的 `operator()`），并传回一个“可被转换为 **AbstractProduct***”的指针。先前演示的具体实作码中，**ProductCreator** 是个简单的函数指针。如果我们只需要

通过 `new` 来生成对象 (这是最常见的情况), 那么这就足够了。因此, 我们选择

```
AbstractProduct* (*)()
```

作为 `ProductCreator` 的缺省型别。这个型别看起来有点令人费解, 因为它没有名称。如果在括号中的星号后面加上一个名称:

```
AbstractProduct* (*PointerToFunction)()
```

那就比较清楚地表示: 我是个函数指针, 我所指向的函数没有任何参数, 传回的是一个指向 `AbstractProduct` 的指针。如果这对你仍然陌生, 或许你需要参阅第 5 章, 那儿提供了函数指针的讨论。

说到那一章, 顺便提一句: 有个非常有趣的 `template` 参数可以作为 `ProductCreator` 传递给 `Factory`, 那就是 `Functor<AbstractProduct*>`。如果选择这个参数, 你将获得很大的灵活性: 你可以通过“一般函数”或“成员函数”或“仿函数”, 并对它们“绑定适当参数”, 用以生成对象。胶合码 (glue code) 由 `Functor` 提供。

现在, `Factory class template` 的声明看起来像这样:

```
template
<
    class AbstractProduct,
    class IdentifierType,
    class ProductCreator = AbstractProduct* (*)(),
    template<typename, class>
        class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;
```

终于我们的这个 `Factory class template` 可以使用了。

8.7 Clone Factories (克隆工厂、翻制工厂、复制工厂)

尽管制造宇宙战士克隆人的基因工厂 (genetic factories) 是一个令人极度恐慌的想象, 但是大多数情况下, 克隆 (clone) C++ 对象却是一件有益无害的事。这儿的目標与先前稍有不同: 我们不再两手空空产生对象, 我们有一个指向多态对象的指针, 希望产生一些和它完全相同的拷贝。但由于不知道这个多态对象的确切型别, 所以无法确切知道将产生的是什么样的新对象。这正是问题所在。

我们手上确实有个对象, 因此可以运用经典的多态特性。克隆对象的常用手法是在 `base class` 中声明一个虚函数 `Clone`, 并让每一个 `derived class` 改写它。这里有一个例子, 使用先前的几何图形继承体系:

```
class Shape
{
```

```

public:
    virtual Shape* Clone() const = 0;
    ...
};

class Line : public Shape
{
public:
    virtual Line* Clone() const {
        return new Line(*this);
    }
    ...
};

```

注意此处 `Line::Clone()` 传回的不是 `pointer to Shape`。也就是说，我们运用了 C++ 所谓的“协变式返回型别 (covariant return type)”特性。基于这一特性，在改写的虚函数中，你可以传回 `pointer to derived class`，而不必一定传回 `pointer to base class`。从现在起，这个手法要持续运用，对于加入继承体系的每一个 `class`，你都必须实作一个类似的 `Clone()`。它们的内容都差不多：如果要生成一个 `Polygon`，就传回 `new Polygon(*this)`，依此类推。

这一手法是有效的，但有几个主要缺点：

- 如果 `base class` 并未被设计为可克隆（也就是说它没有声明诸如 `Clone()` 的虚函数），而且又不可修改，那么这一招就无从运用。如果你所编写的程序用上一个 `class library`，而你需从后者提供的 `base class` 派生自己的 `classes`，这种情况下上述手法就无法使用（译注：因为这个第三方 `class library` 中的 `base class` 不一定被设计为可克隆或可修改）。
- 即使所有 `classes` 都可修改，上述手法也有很大约束。如果你忘记在某个 `derived class` 中实作 `Clone()`，编译器不会帮你检测出来，因而造成执行期的奇怪行为（甚至是有害的行为）。

第一点很明显；我们来谈谈第二点。假设你从 `Line` 派生出一个 `DottedLine`，但忘记改写 `DottedLine::Clone()`。现在假设你有一个 `Shape` 指针，但实际指向一个 `DottedLine`，然后你对它调用 `Clone()`：

```

Shape* pShape;
...
Shape* pDuplicateShape = pShape->Clone();

```

此时 `Line::Clone()` 将被调用，传回一个 `Line`。这是很不幸的情况，因为你原本以为 `pDuplicateShape` 将因此具备和 `pShape` 相同的动态型别，事实却非如此。这会带来很多问题，小至绘出意想不到的线形，大至造成程序崩溃。

没有什么坚实的办法可以减轻第二个问题。你无法以 C++ 告诉大家：“我定义了这个函数，我要求所有直接或间接继承它的 `classes` 都必须改写该函数”。在任何 `Shape derived classes` 中你都必须肩负“改写 `Clone()`”这一麻烦而重复的任务；如果你不这样做，那就等着瞧！

如果你不介意将手法复杂化一些，你可以施行一次尚可承受的执行期检验：将 `Clone()` 声明为 `public non-virtual` 函数，并在其内部调用一个 `private virtual` 函数，例如 `DoClone()`，然后检查动

态型别是否相等。实际代码比文字说明清楚得多：

```
class Shape
{
    ...
public:
    Shape* Clone() const //nonvirtual
    {
        // delegate to DoClone
        Shape* pClone = DoClone();
        // Check for type equivalence
        // (could be a more sophisticated test than assert)
        assert(typeid(*pClone) == typeid(*this));
        return pClone;
    }
private:
    virtual Shape* DoClone() const = 0; // private
};
```

唯一的缺点是，你不再能够使用协变式返回型别 (covariant return type)。

Shape 派生类应该总是改写 DoClone(), 并让它保持 private, 使客户无法调用之, 只留 Clone() 唱独角戏。客户只使用 Clone(), 在其内进行执行期检测。不过, 正如你理应想到的, 某些编程错误 (例如改写 Clone() 或是将 DoClone() 声明为 public) 还是会发生。

别忘了, 无论如何, 如果你无法改变继承体系 (那是封闭的) 中的所有 classes, 而且如果它们并未被设计为可克隆 (clonable), 你就没有任何机会施展这一手法。很多时候这都足以成为一个让人放弃的理由, 所以我们得另谋他法。

这种情况下, 特殊的 object factory 或许可以带来帮助。其解法不存在前面提到的两个问题, 但必须牺牲一些效率。这种特殊解法不使用虚调用 (virtual call), 而是改采用一个 “map 查询动作” 加上一个 “经由函数指针的调用”。我们知道, 任何程序中的 classes 数目都不会很庞大 (它们都是人写出来的不是吗?), 因此, map 不会很庞大, 效率上的损失也就不显著。

这个手法源于以下思想: 在一个 clone factory 中, “型别标识符” 和 “产品” 有着相同的型别。你接收 “待复制物” 并将它视为 “型别标识符”, 再将该 “型别标识符” 的拷贝 (一个新对象) 当做 “输出” 传出去。精确地说它们其实并非隶属相同型别: cloning factory 的 IdentifierType 是一个 pointer to AbstractProduct。真正的交易是, 你传递一个 pointer to clone factory, 得到一个 pointer to cloned object。

但 map 中的 key 应该是什么呢? 它不该是个 pointer to AbstractProduct, 因为 map 中的元素数目不需要像我们拥有的对象那样多。一个 “待复制物” 型别只需要一个 map 元素来对映即可, 这将我们再次带到 std::type_info class 面前。 “要求工厂产生新对象” 时所传递的型别标识符, 和 “存储于相关 map 内” 的型别标识符是不同的, 这使我们不可能复用 (reuse) 迄今为止所写的代码。另一个结果是, 产品生产者 (product creator) 如今需要一个参数: 一个指向 “待

复制物”的指针。先前“从无到有”(而非复制)的工厂中,产品生产者(*product creator*)并不需要这个参数。

让我们回顾一下。clone factory 得到一个 pointer to AbstractProduct。它将 typeid 操作符施行于被指对象身上,得到一个 reference to std::type_info 对象。然后 clone factory 在其 private map 中查询该对象(std::type_info 的 before() 成员函数为“std::type_info 对象集”引入了次序关系,使我们有机会运用 map 执行快速查找)。如果 map 之中没有找到相应元素,将有一个异常被抛出去。如果找到相应元素,产品生产者(*product creator*)会被调用,并接受客户传入的一个 pointer to AbstractProduct。

我们手上已有 Factory class template,所以要实作出 CloneFactory class template 可谓小菜一碟(你可以在 Loki 中找到其实作码)。它有一些不同之处,还有一些新成分:

- CloneFactory 用的是 TypeInfo, 而不是 std::type_info。第2章曾对 TypeInfo class 作过讨论,那是包装于 pointer to std::type_info 之上的一个 class,用来定义合适的初始化操作、operator=, operator==, operator<, 这些操作和操作符对 map 都是必要的。第一个操作符委派(delegates)给 std::type_info::operator==, 第二个操作符委派给 std::type_info::before()。
- 不再需要 IdentifierType, 因为标识符的型别是隐式的(implicit)。
- template 参数 ProductCreator 缺省为 AbstractProduct* (*) (AbstractProduct*)。
- IdToProductMap 如今是 AssocVector<TypeInfo, ProductCreator>。

CloneFactory 大致实作如下:

```
template
<
    class AbstractProduct,
    class ProductCreator = AbstractProduct* (*) (AbstractProduct*),
    template<typename, class>
        class FactoryErrorPolicy = DefaultFactoryError
>
class CloneFactory
{
public:
    AbstractProduct* CreateObject(const AbstractProduct* model);
    bool Register(const TypeInfo&, ProductCreator creator);
    bool Unregister(const TypeInfo&);
private:
    typedef AssocVector<TypeInfo, ProductCreator> IdToProductMap;
    IdToProductMap associations_;
};
```

对于封闭的(也就是你无法修改的)class 继承体系来说,如果要复制其中对象,CloneFactory class template 是一个完整方案。它的简易性和高效率基于先前各小节对概念的澄清,以及 C++

通过 `typeid` 和 `std::type_info` 提供的执行期类型信息 (RTTI)。如果没有 RTTI, 实作 `clone factory` 时我们的处境将更为艰难——艰难到将它们组装起来并没有多大意义。

8.8 通过其他泛型组件来使用 Object Factories

第 6 章介绍的 `SingletonHolder` class, 用来向你的 classes 提供特定服务。`factory` 天生具备全局性, 因此, 通过 `SingletonHolder` 来使用 `Factory` 是很自然的事。借助 `typedef`, 二者可以轻易结合, 例如:

```
typedef SingletonHolder< Factory<Shape, std::string> > ShapeFactory;
```

当然, 你可以为 `SingletonHolder` 或 `Factory` 添加引数, 以便进行不同的取舍, 或选择不同的设计策略, 但它们全数在单一某处完成。从现在起你可以在单一某处将一组重要的设计决策隔离出来, 并在程序的任何地点使用 `ShapeFactory`。上面所演示的简单型别定义中, 你可以对 `factory` 的工作方式和 `singleton` 的工作方式加以选择, 进而开发出二者之间的所有组合。靠着单一行声明式, 你就可以指挥编译器为你生成正确的代码 (此外再无其他影响), 这就像是执行期间通过各种参数调用某个函数, 从而以不同方式执行某个动作一样。由于我们的一切都发生在编译期, 所以重点更多摆在设计决策上, 而非摆在执行期行为。当然, 执行期行为也会受到影响, 但其方式更具全局性、更隐蔽。编写“一般”代码时, 你指定的是“执行期将发生什么事”, 然而写下上述型别定义时, 你指定的是“编译期将发生什么事”——有点像是“调用编译期代码生成函数, 并为这些函数传递引数”。

如同本章一开始所提, 一种有趣的组合是: 通过 `Functor` 来使用 `Factory`:

```
typedef SingletonHolder
<
    Factory
    <
        Shape, std::string, Functor<Shape*>
    >
>
ShapeFactory;
```

这么一来, 借助 `Functor` (第 5 章: 为了实作它我们费尽了心血) 的威力, 你在生成对象时便有了很大的灵活度。现在, 只要为 `Factory` 注册各种不同的 `Functors`, 你就可以用几乎任何可以想象的方式来产生 `Shapes`, 并且这整个东西是一个 `Singleton`。

8.9 摘要

在使用多态特性（polymorphism）的程序中，object factories（对象工厂）是其重要组成部分。当对象型别不可知，或虽可知却无法通过语言构件来使用时，object factories 有助于产生那些对象。

object factories 主要用于面向对象程序框架（OO frameworks）和程序库（libraries），以及各种对象持久（persistence）和串流处理（streaming）的设计中。对于后一种情况，我们通过一个具体实例做了深入分析。我们所讨论的解决方案实质上是将“型别转换（switch of type）”散布于多个实作文件中，从而获得低耦合性。虽然 factories 依然是对象生成的中心机构，但它无需收集继承体系中的所有静态型别相关信息。相反地，每个型别有责任将自己注册给 factories。这反映出错误法门和正确法门之间本质上的不同。

在 C++ 中，型别信息无法很方便地在执行期间传递。这是 C++ 所属之语言家族的基本特征。正因如此，用来表示型别的“型别标识符”才有用武之地。型别标识符和 *creator* 对象有关；所谓 *creator* 对象是个可调用体（callable entity）——第 5 章讨论泛化仿函数（函数指针或仿函数）时曾对此作过介绍。基于这些思想，我们具体实现了一个 object factories，并将它泛化为一个 class template。

最后，我们讨论了 clone factories，这是一种可以复制“多态对象”的工厂。

8.10 Factory Class Template 要点概览

- Factory 声明如下：

```
template
<
    class AbstractProduct,
    class IdentifierType,
    class ProductCreator = AbstractProduct* (*)(),
    template<typename, class>
        class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;
```

- 如果你想为某一继承体系提供 object factories，上述声明中的 AbstractProduct 应指定为该继承体系中的 base class。
- IdentifierType 是一种所谓的 “cookie” type（昵称），用来表示继承体系中的某个型别。IdentifierType 必须是有序型别（ordered type），使之得以存储于 std::map。常用的标识符是字符串和整数。
- ProductCreator 是用来生成对象的一个可调用体（callable entity）。这个型别必须支持 operator()，并且不带参数，传回一个 pointer to AbstractProduct。ProductCreator 对象总是和一个型别标识符注册在一起。
- Factory 实作出以下基本操作。

```
bool Register(const IdentifierType& id, ProductCreator creator);
```

以上函数注册 *creator* 时必须通过型别标识符。如果注册成功就传回 `true`，否则就传回 `false`（表示已有一个以同一型别标识符注册成功的 *creator*）。

```
bool Unregister(const IdentifierType& id);
```

以上函数为某一型别标识符取消相应的 *creator* 注册。如果该型别标识符以前的确注册过，就传回 `true`。

```
AbstractProduct* CreateObject(const IdentifierType& id);
```

以上函数在内部 `map` 中查询型别标识符。如果找到，则针对它调用相应的 *creator*，并传回结果。如果没找到，则传回 `FactoryErrorPolicy<IdentifierType, AbstractProduct>::OnUnknownType()` 的运行结果。`FactoryErrorPolicy` 缺省实作码会抛出一个型别为 `Exception` 的异常；`Exception` 是个嵌套（nested, inner）类。

8.11 Clone Factory Class Template 要点概览

- `CloneFactory` 声明如下：

```
template
<
    class AbstractProduct,
    class ProductCreator =
        AbstractProduct* (*)(Const AbstractProduct*),
    template<typename, class>
        class FactoryErrorPolicy = DefaultFactoryError
>
class CloneFactory;
```

- 如果你想为某一继承体系提供 clone factory，上述声明中的 `AbstractProduct` 应指定为该继承体系中的 `base class`。
- `ProductCreator` 的角色是复制经由参数传入的对象，并传回一个指向复制品（clone）的指针。
- `CloneFactory` 实作出以下基本操作。

```
bool Register(const TypeInfo&, ProductCreator creator);
```

以上函数根据一个 `TypeInfo`（此型别接受一个 `std::type_info` 隐式转换构造函数）对象注册 *creator*。注册成功就传回 `true`，否则传回 `false`。

```
bool Unregister(const TypeInfo& typeInfo);
```

以上函数为某一型别取消相应的 *creator* 注册。如果该型别曾经注册过，函数传回 `true`。

```
AbstractProduct* CreateObject(const AbstractProduct* model);
```

以上函数在内部 `map` 中查询 `model` 的动态型别。如果找到，则调用该型别标识符所对应的 *creator*，并传回结果。如果没找到，则传回 `FactoryErrorPolicy<OrderedTypeInfo, AbstractProduct>::OnUnknownType()` 的运行结果。

9

Abstract Factory

抽象工厂

本章讨论设计模式 Abstract Factory (Gamma 等, 1995) 之泛型实现。Abstract factory (抽象工厂) 是一个接口, 用来生成一族系“相互关联”或“相互依赖”的多态对象 (polymorphic objects)。

Abstract factory 是一个重要的“架构型组件 (architectural component)”, 它可以保证在整个系统中产生出正确的具象对象 (concrete objects)。你肯定不会希望看到 FunkyButton 出现在 ConventionalDialog 上; 你可以运用 Abstract Factory 模式, 保证 FunkyButton 只出现在 FunkyDialog 上。要达到这一点, 你只需控制一小段代码, 应用程序的其余部分便自然会和抽象型别 Dialog 及 Button 打交道。

阅读本章之后, 你将能够:

- 了解 Abstract Factory 模式的应用领域
- 知道如何定义和实现 Abstract Factory 组件
- 知道如何使用 Loki 提供的泛型 Abstract Factory 工具, 并知道如何扩充它以适应你的需要

9.1 Abstract Factory 扮演的体系结构角色 (Architectural role)

假设你正从事一项令人羡慕的工作——设计一个“格斗”游戏, 就像 Doom 或 Quake 那样。

你希望吸引一般纳税人玩你的游戏, 所以我为游戏提供一个初等级别。在初等级别中, 敌方士兵反应迟钝, 怪兽步履缓慢, 就连超级怪兽也十分温和。

你也希望吸引发烧级玩家玩你的游戏, 所以你提供了一个高难度级别。在此级别中, 敌方士兵一秒钟可以连发三枪, 并且是空手道九段; 怪兽狰狞狡诈, 时而还会出现真正可怕的超级怪兽。

要模拟这样一个恐怖世界, 一种可能的做法是: 定义一个 **Enemy base class**, 并从它派生出三个精确接口: **Soldier**, **Monster**, **SuperMonster**。然后再从这些接口派生出 **SillySoldier**, **SillyMonster** 和 **SillySuperMonster** 用于初等级别, 以及 **BadSoldier**, **BadMonster** 和 **BadSuperMonster** 用于高难度级别。最终的继承体系如图 9.1 所示。

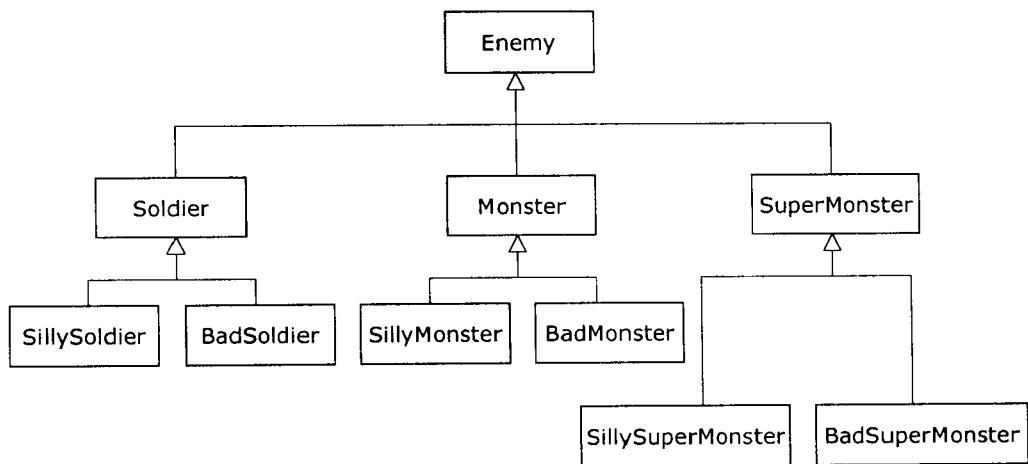


图 9.1 这样的继承体系适合前述“提供两种级别（初等级别和高难度级别）”的游戏
（silly: 愚蠢的，bad: 厉害的）

值得注意的是, 在你的游戏中, `BadSoldier` 具现体 (instantiation) 和 `SillyMonster` 具现体不会同时存在。是的, 那没有意义, 玩家要么就和 `SillySoldiers`, `SillyMonsters`, `SillySuperMonsters` 玩初级游戏, 要么就和 `BadSoldiers`, `BadMonsters`, `BadSuperMonsters` 进行艰苦战斗。

这两大类型的 `types` 构成了两个族系 (family)。游戏期间你一定会使用这两个族系中的某一族对象, 但绝不会同时使用它们。

如果能够确保这种一致性就太好了。否则万一程序稍有疏忽, 那些正在痛击 `SillySoldiers` 的游戏菜鸟们一扭头遇上一个 `BadMonster`, 一败涂地, 你也只好准备兑现退款承诺。

只需小心一次, 当然比必须小心一百次好; 所以你大概会将所有游戏对象的生成函数集中到一个接口上, 像这样:

```
class AbstractEnemyFactory
{
public:
    virtual Soldier* MakeSoldier() = 0;
    virtual Monster* MakeMonster() = 0;
    virtual SuperMonster* MakeSuperMonster() = 0;
};
```

然后你便可以针对不同的难度级别, 实作出一个具体的 `enemy factory` (敌军工厂), 这个 `factory` 将根据游戏策略规定去生成 `enemies`。

```
class EasyLevelEnemyFactory : public AbstractEnemyFactory
{
public:
    Soldier* MakeSoldier()
    { return new SillySoldier; }
    Monster* MakeMonster()
    { return new SillyMonster; }
    SuperMonster* MakeSuperMonster()
    { return new SillySuperMonster; }
};

class DieHardLevelEnemyFactory : public AbstractEnemyFactory
{
public:
    Soldier* MakeSoldier()
    { return new BadSoldier; }
    Monster* MakeMonster()
    { return new BadMonster; }
    SuperMonster* MakeSuperMonster()
    { return new BadSuperMonster; }
};
```

最终, 你可以通过合适的具象类来初始化 `pointer to AbstractEnemyFactory`:

```
class GameApp
```

```

{
    ...
    void SelectLevel()
    {
        if (user chooses the Easy level) {
            pFactory_ = new EasyLevelEnemyFactory;
        }
        else {
            pFactory_ = new DieHardLevelEnemyFactory;
        }
    }
private:
    // Use pFactory_ to create enemies
    AbstractEnemyFactory* pFactory_;
};

```

这个设计的优点是，“生成及正确匹配 *enemies*”的所有细节都由 *AbstractEnemyFactory* 的两个实作版本来管理。由于应用程序只是将 *pFactory_* 视为唯一的对象生成者，因此，一致性（consistency）便通过设计获得了保证。这是 *Abstract Factory* 模式的一个典型应用。

Abstract Factory 模式要求，所有对象族系的生成函数都要集中于唯一接口。所以，针对“待生成物”的每一个族系，你都必须为这唯一接口提供一份实作版本。

Abstract Factory 接口中声明的产品型别（亦即本例之 *Soldier, Monster, SuperMonster*），被称为 *abstract products*（抽象产品）。实作版本中实际生成的产品型别（亦即本例之 *SillySoldier, BadSoldier, SillyMonster* 等等），则被称为 *concrete products*（具体产品）。如果你已阅读过第 8 章，这些术语对你应该是很熟悉了。

Abstract Factory 的主要缺点是，它对型别的需求很强烈：*abstract factory base class*（本例中的 *AbstractEnemyFactory*）必须知晓每一个待生成的 *abstract product*。此外，至少在刚刚提供的那个实作版本中，每一个 *concrete factory class* 都与它将生成的 *concrete products* 相依。

运用第 8 章介绍的技巧，你可以降低这种依存性。在第 8 章，当生成具体对象时，你需要知道的并不是对象的型别，而是其型别标识符（例如一个 *int* 或 *string*）。如此一来依存性就弱得多。

但是，依存性降得愈多，对型别信息的了解就愈少，因而在你的设计中，型别安全会被更大幅度地降低。这是“更多的型别安全”和“更少的依存性”两难情况的又一实证；在 C++ 中，这一经典的两难情况会经常出现。

一如既往，想要得到正确解法，你必须权衡利弊。你应该做出最适合你的需求的抉择。这儿有一条经验法则：尽可能使用静态模型（static model），必要时才仰赖动态模型（dynamic model）。

以下数节我提供了一个 *Abstract Factory* 的泛型实现，这份实作品展现出令人注目的特性：它降低静态依存性，而又无损型别安全。

9.2 一个泛化的 Abstract Factory 接口

一如第 3 章提示，有了 `typelist` 这一利器，实现泛化 Abstract Factories 就如同灌篮一样酣畅有力。这一节我将讲述如何借助 `typelists` 来定义一个泛化的 Abstract Factory 接口。

上一节演示的例子是 Abstract Factory 模式的一个典型应用。让我们回顾一下：

- 你定义了一个抽象类（亦即 abstract factory class），它为每一种 *product types* 都提供一个纯虚函数，其中与型别 `T` 对应的虚函数通常传回 `T*`，函数名称则通常为 `CreateT`, `MakeT` 或类似名称。
- 你定义了一个或多个具体工厂（concrete factories），它们实作 abstract factory 所定义的接口。然后你实作每一个成员函数，用来为派生型别生成新对象——通常是利用 `new` 操作符。这一切似乎再简单不过，但随着这个 abstract factory 生成的产品数量增多，代码会变得愈来愈难维护。此外，你随时可能决定添加一份实作品，它也许使用不同的空间分配方式，或使用一个 `prototype` 对象。

泛化的 Abstract Factory 将带给你帮助，但它必须有足够的灵活性，能够轻松适应一些事物或情况，例如采用定制型分配器（custom allocators）或传递引数给构造函数等等。

回顾第 3 章的 `GenScatterHierarchy` class template。它会“以某个 `typelist` 中的每一个型别作为 `template` 参数”，将用户提供的 `base template` 逐一具现出来。通过这样的结构，`GenScatterHierarchy` 的最终具现体继承了“用户提供之 `template`”的所有具现体（见图 3.2）。换句话说，如果你有一个 `Unit template` 和一个 `TList typelist`，那么 `GenScatterHierarchy <TList, Unit>` 就会继承 `Unit<T>`，其中 `T` 是 `TList` 内含的任一型别。

`GenScatterHierarchy` 对于 Abstract Factory 接口的定义非常有用。你可以先定义一个接口，用来生成某种类型的对象，然后通过 `GenScatterHierarchy` 将该接口应用至多个型别。

假设 `T` 为通用型别（generic type），那么可用以生成 `T` 对象的所谓 “unit” 接口将像下面这样：

```
template <class T>
class AFUnit      // 译注：Loki 源码中命名为 AbstractFactoryUnit
{
public:
    virtual T* DoCreate(Type2Type<T>) = 0;
    virtual ~AFUnit() {}
};
```

这个小小的 `template` 看上去极为规范，包括它拥有一个虚析构造函数³⁴。但那个 `Type2Type` 又做什么用呢？第 2 章告诉我们，`Type2Type` 是个简单的 `template`，其唯一用途是消除重载函数的歧义（模棱两可，ambiguity）。好的！但带有歧义的函数又在哪儿呢？`AFUnit` 只定义了一个

³⁴ 第 4 章详细讨论了虚析构造函数（virtual destructors）。

DoCreate()函数啊！你很快会看到，同一个继承体系中有数个 AFUnit 具现体；对于所生成的各个 DoCreate() 重载函数，Type2Type<T> 有助于消除歧义。

泛型 AbstractFactory 接口结合运用 GenScatterHierarchy 和 AFUnit 如下：

```
template
<
    class TList,
    template <class> class Unit = AFUnit
>
class AbstractFactory : public GenScatterHierarchy<TList, Unit>
{
public:
    typedef TList ProductList;
    template <class T> T* Create()
    {
        Unit <T>& unit = *this;
        return this->DoCreate(Type2Type<T>());
    }
};
```

啊哈，正因为 Type2Type 发挥了作用，Create() 才知道该调用哪一个 DoCreate() 函数。让我们分析一下，当你输入以下代码后，会发生什么事：

```
// Application code
typedef AbstractFactory
<
    TYPELIST_3(Soldier, Monster, SuperMonster)
>
AbstractEnemyFactory;
```

正如第3章所言，AbstractFactory template 会生出图 9.2 所示的继承体系。AbstractEnemyFactory 会多重继承 AFUnit<Soldier>，AFUnit<Monster> 和 AFUnit<SuperMonster>。每一个具现体都定义了一个纯虚函数 Create()，所以 AbstractEnemyFactory 有三个 Create() 重载函数。简言之，AbstractEnemyFactory 和前一节定义的同名抽象类几乎是一样的。

AbstractFactory 的 template 成员函数 Create() 是一个分派器 (dispatcher, 发送器)，它将生成请求 (creation request) 分派 (发送) 给相应的 base class:

```
AbstractEnemyFactory* p = ...;
Monster* pOgre = p->Create<Monster>();
```

这个自动生成的版本有一个重要优点：AbstractEnemyFactory 是一个高度“粒度化” (granular) 接口。你可以将一个 reference to AbstractEnemyFactory 自动转换为一个 reference to AbstractFactory<Soldier> 或 AbstractFactory<Monster> 或 AbstractFactory<SuperMonster>。这么一来你就只需要将这个 factory 的某个小小单元传给程序各部分。假

设某个模块（例如 `Surprises.cpp`）只需生成 `SuperMonsters`，你可以藉由 `pointer to- 或 reference to- AbstractFactory<SuperMonster>` 来和该模块通讯，于是 `Surprises.cpp` 就不会和 `Soldier` 及 `Monster` 之间产生耦合（couple）关系。

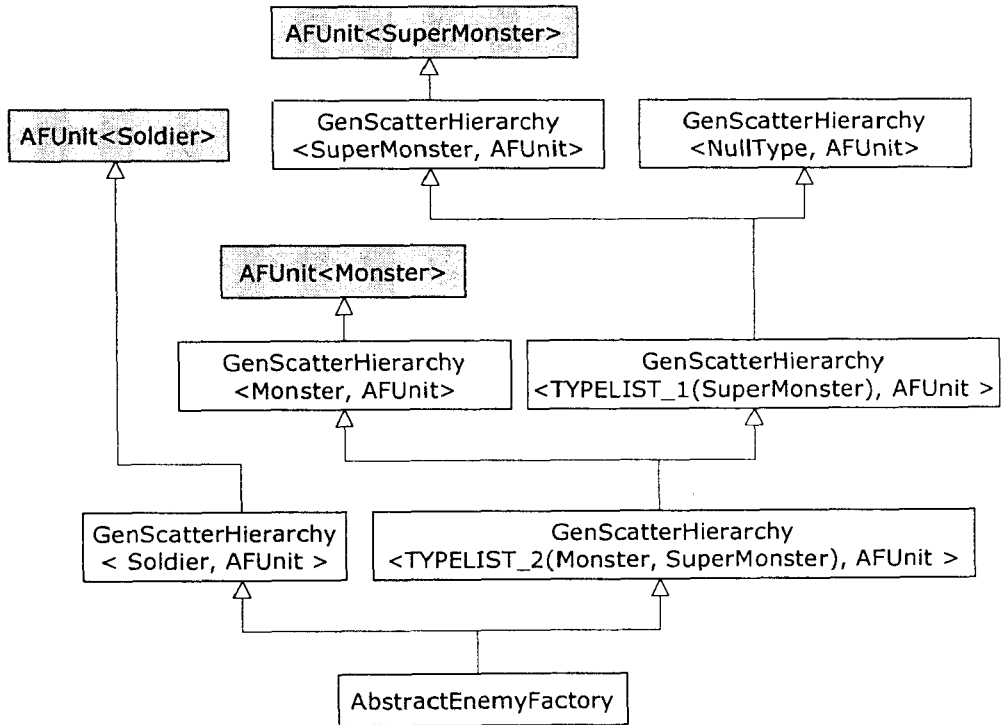


图 9.2 `AbstractEnemyFactory` 所产生的 class 继承体系

利用 **AbstractFactory** 的颗粒性（granularity）得以降低原本会影响设计模式 **Abstract Factory** 的耦合度。你获得了“去耦合”（decoupling）的好处，又无需牺牲 **abstract factory** 的接口安全，因为只有接口才具备颗粒性（granularity），实作码并不具备。

这就将我们带到接口的“实作”主题。自动生成的 **AbstractEnemyFactory** 所具备的第二个重要优点是：你可以将实作自动化。

9.3 实作出 Abstract Factory

现在我们已经定义好了接口（interface），应该设法让实作（implementation）愈简单愈好。

如果定义接口时使用了 **typelists**，那么在建造 **AbstractFactory** 的泛型实作时，一个很自然的方法是使用 *concrete products*（具体产品）的 **typelists**。事实上，建造一个“初等级别”的 **concrete factory** 应该像下面这么简单：

```
// Application code
typedef ConcreteFactory
<
    // The abstract factory to implement
    AbstractEnemyFactory,
    // The policy for creating objects
    // (for instance, use the new operator)
    OpNewFactoryUnit,
    // The concrete classes that this factory creates
    TYPELIST_3(SillySoldier, SillyMonster, SillySuperMonster)
>
EasyLevelEnemyFactory;
```

ConcreteFactory class template 的三个（目前假设的）引数已经提供足够信息，用来实现一个完整的 **factory**：

- **AbstractEnemyFactory** 提供的是“欲实现的 **abstract factory** 接口”，并隐式提供了一组产品（products）。
- **OpNewFactoryUnit** 是个 **policy**，规定如何实际生成对象。第1章曾经详细介绍 **policy class** 的概念。
- **typelist** 提供了“这个 **factory** 所要生成的 **concrete classes**”的集合。这个 **typelist** 中的每一个具象型别都对应于 **AbstractFactory** 的 **typelist** 中具有相同索引的抽象型别。例如，**SillyMonster**（索引1）是 **Monster**（在 **AbstractEnemyFactory** 的定义中具有相同索引）的具象型别。

搭建好 **ConcreteFactory** 框架之后，我们来考虑如何实现它。简单的数学知识就足以让我们做出这样的结论：定义多少个纯虚函数，就有多少个纯虚函数需要改写（否则我们将无法具现化 **ConcreteFactory**；而根据定义，**ConcreteFactory** 是准备投入实际使用的）。所以，**ConcreteFactory** 应该继承 **OpNewFactoryUnit**（它负责实作 **DoCreate**），后者应配合 **typelist** 中的每一个型别各自产生一个具现体。

在这里，身为 `GenScatterHierarchy` 的补充和配对（见第 3 章），`GenLinearHierarchy` class template 可以起很好的作用，因为它可以为我们管理“生成具现体”的所有细节。

`AbstractEnemyFactory` 必须是继承体系的根源（root）。所有的 `DoCreate()` 函数实作码和最终的 `EasyLevelEnemyFactory` 都必须从它派生。`OpNewFactoryUnit` 的每一个具现体都会改写 `AbstractEnemyFactory` 所定义的三个 `DoCreate()` 纯虚函数中的一个。

接下来，让我们定义 `OpNewFactoryUnit`。很明显，这个 class template 以“待生成物之型别”为 template 参数。此外，`GenLinearHierarchy` 还要求 `OpNewFactoryUnit` 接受另一个参数并从它派生（`GenLinearHierarchy` 利用这第二个 template 参数产生图 3.6 所示的串状继承结构）。

```
template <class ConcreteProduct, class Base>
class OpNewFactoryUnit : public Base
{
    typedef typename Base::ProductList BaseProductList;
protected:
    typedef typename BaseProductList::Tail ProductList;
public:
    typedef typename BaseProductList::Head AbstractProduct;
    ConcreteProduct* DoCreate(Type2Type<AbstractProduct>)
    {
        return new ConcreteProduct;
    }
};
```

`OpNewFactoryUnit` 只需做出一些型别上的计算（type calculation），用以确定要实现哪一个 *abstract product*（抽象产品）。

每一个 `OpNewFactoryUnit` 具现体都是“食物链（food chain）”中的一个组成。每一个 `OpNewFactoryUnit` 具现体都会“吃掉”product list 的头部(head)，方法是：改写相应的 `DoCreate` 函数，并将去掉头部的 `ProductList` 向 class 继承体系的下方传递。因此，最顶端的 `OpNewFactoryUnit` 具现体（恰恰位于 `AbstractEnemyFactory` 下方的那一个）会实作出 `DoCreate(Type2Type<Soldier>)`，最底层的 `OpNewFactoryUnit` 具现体则会实作出 `DoCreate(Type2Type<SuperMonster>)`。

让我们扼要说明一下，`OpNewFactoryUnit` 如何体现它在这个食物链中的重要地位。首先它从它的 base class 中引入 `ProductList` 型别，并更名为 `BaseProductList`（看看 `AbstractFactory` 的定义你就会知道，它的确输出的是 `ProductList` 型别）。`OpNewFactoryUnit` 实作的 *abstract product*（抽象产品）是 `BaseProductList` 的头部，因而是 `AbstractProduct` 的定义。最后，`OpNewFactoryUnit` 将 `BaseProductList::Tail` 作为 `ProductList` 再次输出。这样，剩余的 list 便被传递到继承结构的下面。

请注意，`OpNewFactoryUnit::DoCreate()` 并不像相应的纯虚函数那样传回一个 pointer to `AbstractProduct`。相反地，它传回的是一个 pointer to `ConcreteProduct` object。这样的它还

能够被视为“纯虚函数之实作”吗？答案是肯定的，这得感谢所谓“协变式返回型别（covariant return types）”这个 C++ 语言特性。在 C++ 中你可以以“返回型别为 pointer to derived class”的函数改写（override）“返回型别为 pointer to base class”的函数。这大有意义。有了“协变式返回型别”，你要么知道 concrete factory 的确切型别，得到最多的型别信息，要么只知道 factory 的基础型别（base type），得到较少的型别信息。

ConcreteFactory 必须经由 GenLinearHierarchy 生成一个继承结构。其实作码直截了当：

```
template
<
    class AbstractFact,
    template <class, class> class Creator = OpNewFactoryUnit
    class TList = typename AbstractFact::ProductList
>
class ConcreteFactory
    : public GenLinearHierarchy<
        typename TList::Reverse<TList>::Result, Creator, AbstractFact>
{
public:
    typedef typename AbstractFact::ProductList ProductList;
    typedef TList ConcreteProductList;
};
```

GenLinearHierarchy 为 ConcreteFactory 生成的 class 继承体系如图 9.3 所示。

这其中只有一个花样：将 concrete product list 传给 GenLinearHierarchy 时，ConcreteFactory 必须将其反转（reverse）。为什么？嗯，我们得回到图 3.6，那里展示了 GenLinearHierarchy 是如何生成一个继承结构的：它将 typelist 中的型别“自下而上”分布到它的 Unit template 引数上；typelist 中的第一个元素被传给这个 class 继承体系最底部的 Unit 具现体。但是，OpNewFactoryUnit 是以“自上而下”的方式来实作 DoCreate 重载函数的，因此，在将 TList 传给 GenLinearHierarchy 之前，ConcreteFactory 藉由编译期算法 TL::Reverse（见第 3 章）将其作了反转。

此时如果你觉得 AbstractFactory 和 ConcreteFactory 有点让人困惑，或是太复杂，请不要气馁。因为这两个 class templates 都已驾轻就熟地运用了 typelists。Typelist 对你来说可能是个新概念，你得花点时间来熟悉它。如果将 typelist 视为一个黑盒（black-box）——“typelists 之于 types，恰如 list 之于 values”——那么本章的各项实作就显得非常简单了。一旦你真正习惯了 typelists，苍穹近在咫尺。不信？请继续往下阅读。

9.4 一个 Prototype-Based Abstract Factory 实作品

设计模式 Prototype（Gamma 等, 1995）描述了一种“从 prototype（一种原型对象）来生成对象”的方法。你可以通过克隆（clone）prototype 来获得新对象。其要点完全在于：用来克隆的函数（所谓 cloning function）是虚函数。

从第 8 章的详细讨论可以知道，产生多态对象的根本问题也就是“虚构造函数两难问题（virtual constructor dilemma）”：如果完全从无到有产生对象，就得知道待生对象的型别信息；但是多态（polymorphism）却要求不要知道确切型别。

Prototype 模式借着 prototype 对象的运用避开了这一难题。如果你已经有了一个对象（用来当做 prototype），你就可以好好地利用虚函数。虚构造函数的难题对 prototype 本身来说还是存在的，但已经被大大局部化（localized）了。

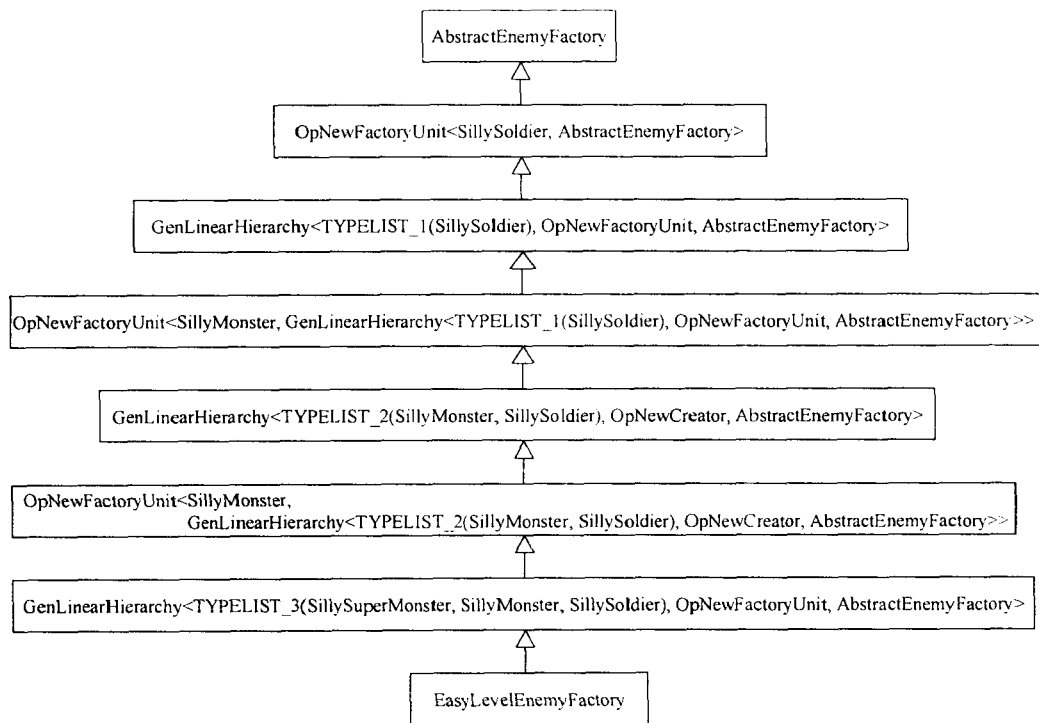


图 9.3 为 EasyLevelEnemyFactory 生成的 class 继承体系

在我们先前那个游戏程序例中，要想建造“敌人”，一个 prototype-based 做法要求：我们得拥有 pointer to- base classes Soldier、Monster、SuperMonster。于是，我们可以写出下面这样的代码³⁵。

³⁵ 警告：这段代码有异常安全（exception safety）方面的问题。修改任务留给读者作为练习。

```

class GameApp
{
    ...
    void SelectLevel()
    {
        if (user chooses Diehard level)
        {
            protoSoldier_.reset(new BadSoldier);
            protoMonster_.reset(new BadMonster);
            protoSuperMonster_.reset(new BadSuperMonster);
        }
        else
        {
            protoSoldier_.reset(new SillySoldier);
            protoMonster_.reset(new SillyMonster);
            protoSuperMonster_.reset(new SillySuperMonster);
        }
    }
    Soldier* MakeSoldier()
    {
        // Each enemy class defines a Clone virtual function
        // that returns a pointer to a new object
        return pProtoSoldier_->Clone();
    }
    ... MakeMonster and MakeSuperMonster similarly defined ...
private:
    // Use these prototypes to create enemies
    auto_ptr<Soldier> protoSoldier_;
    auto_ptr<Monster> protoMonster_;
    auto_ptr<SuperMonster> protoSuperMonster_;
};

```

当然，在现实世界中，接口和实作应该获得更好的切割。这段代码的基本思想是，GameApp 保存 pointer to base enemy classes (即 prototypes)，并利用这些 prototypes 来生成 enemy 对象，办法是：针对这些 prototypes 调用纯虚函数 Clone()。

在 prototype-based Abstract Factory 实作中，我们会为每一种 *product* type 收集一个指针，并由 Clone() 函数产生新的 *products*。

在采用 prototypes 的 ConcreteFactory 中，我们不再需要提供具体型别。在我们那个例子中，建造 SillySoldiers 或 BadSoldiers 不过是小事一桩：只要为 factory object 提供合适的 prototype 就可以了。prototype 的静态型别是 base class (Soldier)。factory 不必知道对象的具体型别；它只是对着合适的 prototype 对象调用纯虚成员函数 Clone()。这就降低了 concrete factory 对具体型别的依赖性。

但是，为了让 GenLinearHierarchy 扩展机制能够正常运作，我们得有个 typelist。请回忆 ConcreteFactory 的声明：

```

template
<
    class AbstractFact,
    template <class, class> class Creator,

```



```

    class TList
>
    class ConcreteFactory;

```

TList 是 concrete product list。在 EasyLevelEnemyFactory 中它也就是 TYPELIST_3 (SillySoldier, SillyMonster, SillySuperMonster)。如果使用 Prototype 模式, TList 将变得无关紧要。但 GenLinearHierarchy 是需要 TList 的, 因为它要为 *abstract product list* 中的每一个 *product* 生成一个 class。怎么办?

这种情况下一个很自然的解法是: 将 *abstract product list* 传给 ConcreteFactory 作为 TList 引数。现在, GenLinearHierarchy 可以生成正确数目的 classes, 并且无需修改 ConcreteFactory 的实作码。

ConcreteFactory 的声明如今变成了:

```

template
<
    class AbstractFact,
    template <class, class> class Creator,
    class TList = typename AbstractFact::ProductList
>
class ConcreteFactory;

```

(请回忆 9.3 节 AbstractFact 的定义, 那儿定义了内部型别 ProductList。)

现在, 让我们来实作 PrototypeFactoryUnit; 这是一个 unit template, 它保存着 prototype, 并调用 Clone。整份实作直截了当, 而且比 OpNewFactoryUnit 还简单。OpNewFactoryUnit 必须维护两个 typelists (*abstract products* 和 *concrete products*), 但 PrototypeFactoryUnit 只需和 *abstract product list* 打交道。

```

template <class ConcreteProduct, class Base>
class PrototypeFactoryUnit : public Base
{
public:
    PrototypeFactoryUnit(AbstractProduct* p = 0)
        : pPrototype_(p)
    {}
    AbstractProduct* GetPrototype() const
    {
        return pPrototype_;
    }
    void SetPrototype(T* pObj)
    {
        pPrototype_ = pObj;
    }
    AbstractProduct* DoCreate(Type2Type<AbstractProduct>)
    {
        assert(pPrototype_);
        return pPrototype_->Clone();
    }
private:
    AbstractProduct* pPrototype_;
};

```

我们在 `PrototypeFactoryUnit` class template 中做了一些假设，这些假设也许适合你的具体情况，也许不适合。首先，`PrototypeFactoryUnit` 并不拥有它的 `prototype`；有时，在对 `prototype` 赋值之前，你可能希望 `SetPrototype` 先删除旧的 `prototype`。第二，`PrototypeFactoryUnit` 使用了一个名为 `Clone` 的函数，用来克隆 *product*。在你的程序中，你可能会使用不同的名称；因为你可能受到另一个程序库的限制，或者你喜欢另一种命名习惯。

如果需要定制你自己的 `prototype-based factory`，你只需写一个和 `PrototypeFactoryUnit` 类似的 `template`。你可以继承 `PrototypeFactoryUnit` 并且只改写必要的函数。例如，假如你想这样实作 `DoCreate()`：如果 `prototype` 指针为 `null`，就让函数返回 `null` 指针。下面是实际写法：

```
template <class AbstractProduct, class Base>
class MyFactoryUnit
    : public PrototypeFactoryUnit<AbstractProduct, Base>
{
public:
    // Implement DoCreate so that it accepts a null prototype
    // pointer
    AbstractProduct* DoCreate(Type2Type<AbstractProduct>)
    {
        return pPrototype_ ? pPrototype_->Clone() : 0;
    }
};
```

让我们回到先前那个游戏实例。为定义一个 `concrete factory`，程序中你只需提供以下代码：

```
// Application code
typedef ConcreteFactory
<
    AbstractEnemyFactory,
    PrototypeFactoryUnit
>
EnemyFactory;
```

概括地说，`AbstractFactory/ConcreteFactory` 这对拍挡为你提供了以下功能：

- 借助 `typelists`，你可以轻松定义 `factories`。
- `AbstractFactory` 继承了它的每一个 `unit`，因此其接口具有很强的颗粒性（*granular*）。你可以将各个 `creator units`（这是指向 `AFUnit<T>` 子对象的一个 `pointer` 或 `reference`）传给不同的模块，从而降低耦合性。
- 通过 `ConcreteFactory`，并提供一个 `policy template` 用来指定生成方式，你就可以实现 `AbstractFactory`。对于静态绑定的 `creation policy`（例如“使用了 `new` 操作符”的 `OpNewFactoryUnit`），你需要传递一个 `typelist`，其中内含这个 `factory` 所将生成的 *concrete products*。
- 一个常见的 `creation policy` 是“运用 `Prototype` 模式”。藉由罐装的 `PrototypeFactoryUnit` class template，你可以轻松地将 `Prototype` 用于 `ConcreteFactory`。

9.5 摘要

Abstract Factory 设计模式提供了一个接口，用来生成一整个族系、彼此相互关联或相互依存的多态对象。经由 Abstract Factory，你可以将 implementation classes 分割为互不相干的多个族系（families）。

借 typelists 和 policy template 之助，我们就有可能实现一个泛型的 abstract factory 接口。typelist 提供的是 product list（*concrete product* 和 *abstract product*）以及 policy templates。

AbstractFactory class template 提供了一个框架，用以定义 abstract factory，并且可以和 AFUnit class template 共同工作。AbstractFactory 需要用户提供一个 *abstract product list*。在内部，AbstractFactory 通过 GenScatterHierarchy（第 3 章）生成一个颗粒化（granular）的接口，这个接口继承了 AFUnit<T>，其中 T 是 abstract product typelist 中的每一个 product type。这种结构给你降低耦合的好机会，你只需将各个 factory unit（工厂单元）传给程序的不同部分。

ConcreteFactory template 用来实现 AbstractFactory 接口：它使用 FactoryUnit policy 以生成对象，并在内部使用 GenLinearHierarchy（第 3 章）。Loki 为 FactoryUnit policy 提供了两个预先定义好的实作品：一个是 OpNewFactoryUnit，运用 new 操作符来生成对象，另一个是 PrototypeFactoryUnit，复制 prototype 以生成对象。

9.6 AbstractFactory 和 ConcreteFactory 要点概览

- AbstractFactory 概貌：

```
template
<
    class TList,
    template <class> class Unit = AFUnit
>
class AbstractFactory;
```

其中 TList 是个 typelist，包含的是这个 factory 将要生成的 *abstract products*；Unit 是个 template，为 TList 中的每一个型别定义了接口。例如以下定义了一个 abstract factory，这个 factory 能够生成 Soldiers、Monsters 和 SuperMonsters。

```
typedef AbstractFactory<TYPELIST_3(Soldier, Monster, SuperMonster)>
    AbstractEnemyFactory;
```

- AFUnit<T> 定义了一个接口，该接口由一个纯虚函数组成，函数原型为 T* DoCreate(Type2Type<T>)。通常你不需要直接调用 DoCreate()，相反地你应该使用 AbstractFactory::Create()。

- **AbstractFactory** 提供一个名为 `Create()` 的 **template** 函数。你可以用 *abstract products* 的任何型别来具现化 `Create()`。例如：

```
AbstractEnemyFactory *pFactory = ...;
Soldier *pSoldier = pFactory->Create<Soldier>();
```

- 为实现 **AbstractFactory** 所定义的接口，Loki 提供一个 **ConcreteFactory** **template**，大致如下：

```
template
<
    class AbstractFact,
    template <class, class> class FactoryUnit = OpNewFactoryUnit,
    class TList = AbstractFact::ProductList
>
class ConcreteFactory;
```

其中 **AbstractFact** 是将被实作出来之“**AbstractFactory** 具现体”，**FactoryUnit** 是 **FactoryUnit** creation policy 之实作品，**TList** 是 *concrete products typelist*。

- **FactoryUnit** policy 实作品将会取用 *abstract product* 及将被生成之 *concrete product*。Loki 定义了两个 **Creator policies**：**OpNewFactoryUnit** (9.3 节) 和 **PrototypeFactoryUnit** (9.4 节)。它们可以良好地为你示范如何定制 **FactoryUnit** policy 实作品。
- **OpNewFactoryUnit** 使用 `new` 操作符来生成对象。如果使用它，你就必须提供一个 *concrete product typelist*，作为 **ConcreteFactory** 的第三参数。例如：

```
typedef ConcreteFactory
<
    AbstractEnemyFactory,
    OpNewFactoryUnit,
    TYPELIST_3(SillySoldier, SillyMonster, SillySuperMonster)
>
EasyLevelEnemyFactory;
```

- **PrototypeFactoryUnit** 保存着 **pointer to abstract product types**，并调用它们各自的 **prototypes** 中的 `Clone()` 成员函数以生成新对象。也就是说，**PrototypeFactoryUnit** 要求每一个 *abstract product T* 都必须定义一个名为 `Clone` 的纯虚成员函数，该函数传回 `T*`，其语义即为“复制对象”。
- 将 **PrototypeFactoryUnit** 用于 **ConcreteFactory** 时，你不需要为 **ConcreteFactory** 提供第三个 **template** 引数。例如：

```
typedef ConcreteFactory
<
    AbstractEnemyFactory,
    PrototypeFactoryUnit,
>
```

10

Visitor

访问者、视察者

本章将讨论“实现设计模式 Visitor (Gamma 等, 1995)”之泛型组件。Visitor 是一种功能强大（此点或有争议）的模式，可以改变 class 设计中的依存性取舍（dependency trade-offs）。

在某些领域，Visitor 可为你带来极大灵活性：你可以为一个 class 继承体系添加虚函数，而无需重新编译它们或它们既有的客户码。但是这种灵活性需要付出代价，它牺牲了设计者看来理所当然的某些功能：如果不重新编译这个继承体系和其既有客户码，你就无法为这个继承体系添加一个 leaf class（枝叶类，最底层类）。因此，Visitor 的应用有其局限：必须是十分稳定（很少需要添加新 classes）并执行繁重处理（经常需要添加新的虚函数）的继承体系。

Visitor 违反程序员的直觉；因此如果想成功运用它，细致的实作和严格的规定必不可少。本章目标是精心设计一个可靠的 Visitor 泛型实作品，尽可能将应用程序开发者的负担降至最低。

阅读本章之后，你将能够：

- 理解 Visitor 的运作原理
- 知道何时应当使用 Visitor 模式，何时不应当使用 Visitor 模式（此点同样重要）
- 了解 visitor 的基本实作法（我指的是 GoF 实作法）
- 知道如何克服“GoF Visitor 实作法”的缺点
- 知道如何将“应该在实现 Visitor 时做出”的一些决定，移转到程序库中
- 掌握一些功能强大的泛型组件：实作那些“专门用来解决你的问题”的 visitors 时，这些组件将对你大有帮助。

10.1 Visitor 基本原理

假设有个 class 继承体系，你想增强它的功能。为了达到目的，你可以增加新的 classes，也可以增加新的虚函数。

增加新的 classes 很容易。你可以派生自某个 leaf class 并实作必要的虚函数：不必修改或重新编译原有的 classes。这是最有效的代码复用（code reuse）了。

增加新的虚函数则很困难。为了能够以多态方式(亦即藉由 `pointer to root class`)操纵对象,你必须为 `root class` 增加虚成员函数,可能还得为继承体系中的其他许多 `classes` 增加虚成员函数。这是个大动作,需要修改 `root class`,而整个继承体系和其用户都依赖这个 `root class`。最终结果是,你必须重新编译一切。

简而言之,从依存性的角度来看,新的 `classes` 容易增加,新的虚成员函数难以增加。

但假设有个继承体系,你很少需要对它添加新 `classes`,却需要经常添加虚函数。这种情况下你拥有一个你不需要的优势:方便添加新 `classes`。你也拥有一个令人烦恼的缺点:难以添加新的虚函数。这正是 Visitor 用武之地。Visitor 以“你不需要的优势”换取“你需要的优势”。有了 Visitor 你便可以方便地为继承体系添加新的虚函数,但同时也造成更难添加新的 `classes`。这项技术带来的执行期成本至少是“一个额外的虚调用”,稍后你会看到这一点。

Visitor 最适用于“对象上的操作截然不同而且互不相干”的时候。此时每一个 `class` 的“状态和操作(`states and operations`)”都变得没有什么关联;将型别从操作中“分离出来”的观点更为合适。这种情况下,将某一概念性操作(`conceptual operation`)的各种实作维护在一起,而不是分布在 `class` 继承体系中,将很有意义。

举个例子,假设你正在开发一个文档编辑器。文档中的组成单元如文字段落、向量图、位图(`bitmap`)等,都是“从一个共同的 `root class` (例如 `DocElement`) 派生出来”的 `classes`。这个文档是个结构化集合(`structured collection`),集合内含 `pointer to DocElements`。你需要遍历整个集合,执行“拼写检查”、“重新排版”、“收集统计数据”等操作。实作这些行为时,理想情况下应该只需添加代码,不需修改既有代码。此外,如果将“用于获取文档统计数据”的所有代码放在一起,维护起来更简便。

“文档统计数据”可能包括字符、非空格字符、单词、图像…的数目。很自然,这些数据得包含在某个名为 `DocStats` 的 `class` 中:

```
class DocStats
{
    unsigned int
        chars_,
        nonBlankChars_,
        words_,
        images_;
    ...
public:
    void AddChars(unsigned int charsToAdd) {
        chars_ += charsToAdd;
    }
    ...similarly defined AddWords, AddImages...
    // Display the statistics to the user in a dialog box
    void Display();
};
```

如果想通过典型的面向对象方式获取统计数据，你通常会在 `DocElement` 中定义一个虚函数，用来处理“统计数据”的收集：

```
class DocElement
{
    ...
    // This member function helps the "Statistics" feature
    virtual void UpdateStats(DocStats& statistics) = 0;
};
```

然后，每一个具体的“文档组成单元”都会以自己的方式定义这个函数。例如，从 `DocElement` 衍生出来的两个 classes `Paragraph` 和 `RasterBitmap`，会像下面这样实作出 `UpdateStats()`：

```
void Paragraph::UpdateStats(DocStats& statistics)
{
    statistics.AddChars(number of characters in the paragraph);
    statistics.AddWords(number of words in the paragraph);
}

void RasterBitmap::UpdateStats(DocStats& statistics)
{
    // A raster bitmap counts as one image
    // and nothing else (no characters etc.)
    statistics.AddImages(1);
}
```

最后，驱动函数如下：

```
void Document::DisplayStatistics()
{
    DocStats statistics;
    for (each DocElement in the document)
    {
        element->UpdateStats(statistics);
    }
    statistics.Display();
}
```

这就相当不错地实现了“统计功能”，但存在以下数个缺点：

- `DocElement` 及其派生类需得取用 `DocStats` 的定义。因此，每次修改 `DocStats`，你都必须重新编译整个 `DocElement` 继承体系。
- “收集统计数据”的各项实际操作，分布在各个 `UpdateStats()` 实作码中。要想调试或增强统计功能，维护者必须查找并编辑多个文件。
- 增加其他类似“收集统计数据”操作时，上述实作技术无法扩充使用。如果想增加一个操作（例如“将字体大小增加一个点”），你得为 `DocElement` 增加另一个虚函数（并承受继之而来的所有挑战）。

为解除 DocElement 对 DocStats 的依存性, 解法之一是将所有操作移转到 DocStats class 中, 由 DocStats 来确认该为每一个具体型别做些什么。这意味着 DocStats 得有一个成员函数 void UpdateStats(DocElement&)。于是整份文档只需遍历所有组成元素, 并对每个元素调用 UpdateStats()。

这个解法可以有效地使 DocStats 对 DocElement 而言变成无形, 但这么一来却变得 DocStats 依存于它所需要处理的每一个 DocElement 具象类。如果对象的继承体系比对象的操作更稳定 (译注: 更不轻易改变), 这种依存关系还不至于令人烦恼。但另一个问题是 UpdateStats() 的实作因而必须仰赖所谓的 "type switch"。只要你查询一个多态对象的具体型别, 并根据该型别的不同而执行不同操作, 那么 "type switch" 就会发生。例如 DocStats::UpdateStats() 一定需要执行一个像这样的 "type switch":

```
void DocStats::UpdateStats(DocElement& elem)
{
    if (Paragraph* p = dynamic_cast<Paragraph*>(&elem))
    {
        chars_ += p->NumChars();
        words_ += p->NumWords();
    }
    else if (dynamic_cast<RasterBitmap*>(&elem))
    {
        ++images_;
    }
    else ...
        add one 'if' statement for each type of object you inspect
}
```

(上述 if 测试句中, p 的定义之所以合法, 仰仗 C++ 中一条不出名的附加规则。在 if 语句中你可以定义并测试一个变量, 该变量的生命期延续至 if 语句及其 else 部分——如果有 else 的话。虽然这不是一个绝对必需的性质, 而且撰写花俏代码并不值得提倡, 但这一功能是专门用来支持 "type switch" 的, 既然如此何不利用这一便利性呢?)

只要看见这样的东西, 马上要敲起思想警钟。"type switch" 决不是理想的解决方案 (第 8 章提供了详尽的论据)。代码如果仰赖 "type switch", 不但难以理解、难以扩充、难以维护, 而且会带来隐伏的“臭虫”。例如你把一个“转型至 base class”的 dynamic_cast 放在一个“转型至 derived class”的 dynamic_cast 之前, 将如何? 第一款测试会成功, 因而第二款测试永远不会成功。多态 (polymorphism) 的目标之一就是要消除问题多多的 "type switch"。

在这里, 使用 Visitor 模式将大有助益。你需要一些以虚拟方式运作的新函数, 但却不想为每一个操作增加一个对应的虚函数。为达目的, 在 DocElement 继承体系中, 你必须实现一个“专用而唯一的跳转虚函数 (unique bouncing virtual function)”, 其作用是将工作“远程传输 (teleports)”到另一个不同的继承体系中。DocElement 继承体系被称为 *visited hierarchy* (受访体系), 操作则隶属于新的 *visitor hierarchy* (出访体系)。

“跳转虚函数”的每一份实作品调用的是 *visitor hierarchy* 中的不同函数——受访型别 (visited types) 就是这样挑选出来的。被“跳转函数”调用的“*visitor hierarchy* 函数”是虚函数——操作 (operations) 就是这样挑选出来的。

以下数行代码演示了这一想法。首先我们定义一个名为 `DocElementVisitor` 的抽象类，它为 `ElementDoc` 继承体系中的每一个对象型别都定义一个对应的操作函数。

```
class DocElementVisitor
{
public:
    virtual void VisitParagraph(Paragraph&) = 0;
    virtual void VisitRasterBitmap(RasterBitmap&) = 0;
    ... other similar functions ...
};
```

接着我们将名为 `Accept()` 的“跳转虚函数”加到 `DocElement` 继承体系中；`Accept()` 接受 `DocElementVisitor&` 参数，并对这个参数调用相应的 `VisitXxx` 函数。

```
class DocElement
{
public:
    virtual void Accept(DocElementVisitor&) = 0;
    ...
};

void Paragraph::Accept(DocElementVisitor& v)
{
    v.VisitParagraph(*this);
}

void RasterBitmap::Accept(DocElementVisitor& v)
{
    v.VisitRasterBitmap(*this);
}
```

现在，`DocStats` 隆重登场：

```
class DocStats : public DocElementVisitor
{
public:
    virtual void VisitParagraph(Paragraph& par)
    {
        chars_ += par.NumChars();
        words_ += par.NumWords();
    }
    virtual void VisitRasterBitmap(RasterBitmap&)
    {
        ++images_;
    }
    ...
};
```

(当然, 现实中, 这些函数定义应该从 `class` 的定义中提取出来, 放进一个单独的源码文件)

这个小例子也说明了 Visitor 的一个缺点: 你并没有真正增加虚成员函数。真正的虚函数可以完全访问其所在对象; 但是在 `VisitParagraph` 内部你只能取用 `Paragraph` 的 `public` 区域。

测试函数 `Document::DisplayStatistics()` 产生一个 `DocStats` 对象, 并以它为参数, 为每一个 `DocElement` 调用 `Accept()`。当 `DocStats` 对象访问各个实际而具体的 `DocElements` 时, 它用的是完全合适的型别——不需要 “type switching”!

```
void Document::DisplayStatistics()
{
    DocStats statistics;
    for (each DocElement in the document)
    {
        element->Accept(statistics);
    }
    statistics.Display();
}
```

让我们分析一下成果。我们增加了一个新的继承体系, 以 `DocElementVisitor` 为根源。这是一个“由操作 (operations) 组成的继承体系”, 其中每个 `class` (例如 `DocStats`) 实际上都代表一个操作。增加新操作变得很简单, 我们只需从 `DocElementVisitor` 派生出一个新 `class` 就行了。`DocElement` 继承体系中的元素都无需修改。

举个例子, 让我们增加一个新操作 `IncrementFontSize`, 以之为辅助函数, 用来实作“增加字型大小”的热键或工具栏按钮 (toolbar button)。

```
class IncrementFontSize : public DocElementVisitor
{
public:
    virtual void VisitParagraph(Paragraph& par)
    {
        par.SetFontSize(par.GetFontSize() + 1);
    }
    virtual void VisitRasterBitmap(RasterBitmap&)
    {
        // nothing to do
    }
    ...
};
```

就这样! 无需修改 `DocElement` 继承体系, 亦无需修改其他操作。你只需要增加一个新 `class`。`DocElement::Accept()` 会跳至 `IncrementFontSize` 对象, 就像它会跳至 `DocStats` 对象一样。形成的 `class` 结构如图 10.1。

请稍事回忆。一般情况下新 `classes` 容易增加, 但新的虚成员函数不容易增加。藉由从 `DocElement`

继承体系跳转至 DocElementVisitor 继承体系，我们将 classes 转为函数：因此，DocElementVisitor 派生物实际上就是“被对象化了的函数”（objectified functions）。Visitor 设计模式就是这样运作的。

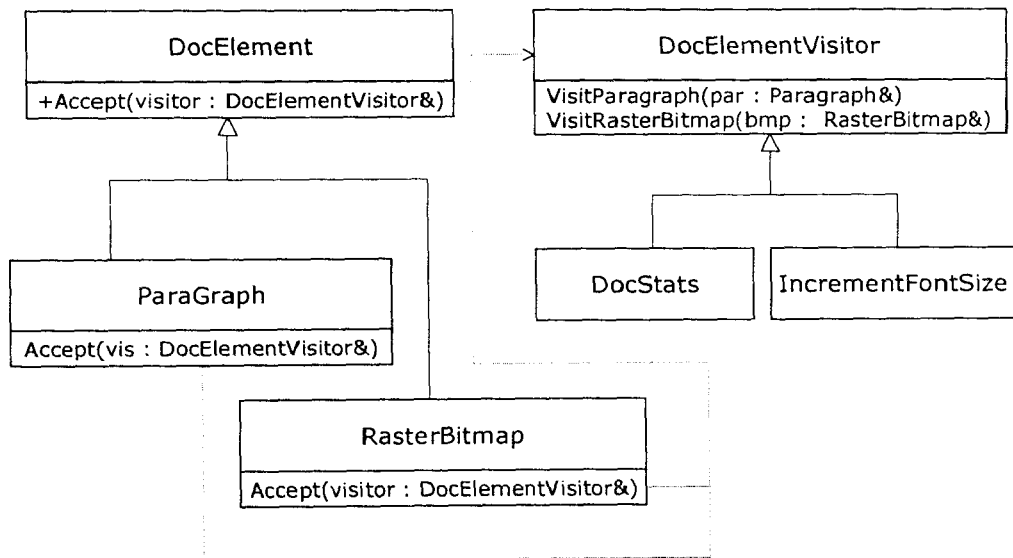


图 10.1 *visitor hierarchy* (右端) 和 *visited hierarchy* (左端)，以及操作的远端传输 (teleported)

10.2 重载 (Overloading) : Catch-All 函数

C++ 重载特性 (overloading) 和 Visitor 模式并没有密切关系, 但是在使用 Visitor 的设计中, 使用或不使用重载, 会对实作带来巨大影响。

在 `DocElementVisitor` 中, 我们为每一种受访型别 (visited type) 定义一个对应的成员函数: `VisitParagraph(Paragraph&)`、`VisitRasterBitmap(RasterBitmap&)`, 等等。这些函数造成了某种赘余, 而且受访元素的名称也被嵌入 (写死于) 函数名称中, 不好。

通常, 赘余能免则免。通过 C++ 重载, 我们可以消除赘余。只需将所有函数命名为 `Visit`, 然后把一切交给编译器即可; 根据传递给 `Visit` 的参数型别, 编译器会判断出应该调用 `Visit` 的哪一份重载。所以 `DocElementVisitor` 的另一种定义像这样:

```
class DocElementVisitor
{
public:
    virtual void Visit(Paragraph&) = 0;
    virtual void Visit(RasterBitmap&) = 0;
    ... other similar functions ...
};
```

所有 `Accept()` 成员函数的定义也将变得更简单。如今它们表现出极好的一致性 (uniformity):

```
void Paragraph::Accept(DocElementVisitor& v)
{
    v.Visit(*this);
}

void RasterBitmap::Accept(DocElementVisitor& v)
{
    v.Visit(*this);
}
```

它们看上去如此相近, 你可能忍不住想把它们提取 (factor) 出来, 放到 base class `DocElement` 中。错! 这种相近只是表象。实际上这些函数大不相同: 在 `Paragraph::Accept()` 中 `*this` 的静态型别是 `Paragraph&`; 在 `RasterBitmap::Accept()` 中 `*this` 的静态型别是 `RasterBitmap&`。正因为藉助于静态型别, 编译器才能判断出应该调用哪一个 `DocElementVisitor::Visit()` 重载版本。如果在 `DocElement` 中实现 `Accept()`, `*this` 的静态型别将是 `DocElement&`, 这并不能为编译器提供所需的型别信息, 所以“提取至 base class”不是个可行方案, 事实上它会使我们的设计失效。

重载的运用引发了一个有趣概念。我们假设, 实作 `Accept()` 时, `DocElement` 的所有派生物都单纯地跳转至 `DocElementVisitor::Visit()`。那么, 我们可以为 `DocElementVisitor` 提供下面这个“catch-all 重载函数”。

```
class DocElementVisitor
{
public:
```

```
... as above ...
virtual void Visit(DocElement&) = 0;
};
```

这份重载版本何时会被调用？如果从 `DocElement` 直接派生一个新的 class，并且在 `DocElementVisitor` 中你并没有为这个 class 提供一个合适的 `visit` 重载函数，那么“重载规则”和“derived class 至 base class 的自动转型”就会起作用。reference to unknown class 会被自动转型为 reference to `DocElement` base class，这个“catch-all 成员函数”就会被调用。如果你没有提供这样一个 catch-all 函数，就会出现编译错误。你可能喜欢这种做法，也可能不喜欢它——取决于你的具体情况。

在这个 catch-all 重载函数中，你可以做很多事情。具体做法可参考（例如）John Vlissides 的著作（Vlissides 1998, 1999）。你可以在其中部署一些防范性措施，或做一些一般性事情，或是求诸“type switch”（经由 `dynamic_cast` 这一非常手段）找出 `DocElement` 的实际型别。

10.3 一份更加精练的实作品：Acyclic Visitor

现在，你决定使用 Visitor 了。让我们务实一点：实际项目中应当如何组织代码？

对前例存在的依存性（dependency）进行分析之后，可以得出以下结论：

- `DocElement` class 定义式若要能够通过编译，必须知晓 `DocElementVisitor`，因为后者出现于 `DocElement::Accept()` 成员函数的标记式（signature）中。啊，好极了，前置声明（forward declaration）可以解决这一问题。
- `DocElementVisitor` class 定义式若要能够通过编译，必须知道（至少）`DocElement` 继承体系中所有的具象类，因为它们名称出现在 `DocElementVisitor` 的 `VisitXxx()` 成员函数中。

这类依存性被称为“循环依存性（cyclic dependency）”。众所周知，循环依存性是维护上的一个瓶颈。`DocElement` 需要 `DocElementVisitor`，`DocElementVisitor` 需要整个 `DocElement` 继承体系。也就是说 `DocElement` 依存于它自己的 subclasses。这是一种“循环名称依存性，*cyclic name dependency*”，也就是说，编译时 class 的定义只依存于其他每个 class 的名称。因此，一种做法是将 classes 合理划分至不同的文件中，像下面这样：

```
// File DocElementVisitor.h
class DocElement;
class Paragraph;
class RasterBitmap;
... forward declarations for all DocElement derivatives ...

class DocElementVisitor
{
    virtual void VisitParagraph(Paragraph&) = 0;
    virtual void VisitRasterBitmap(RasterBitmap&) = 0;
    ... other similar functions ...
};
```

```
// File DocElement.h
class DocElementVisitor;

class DocElement
{
public:
    virtual void Accept(DocElementVisitor&) = 0;
    ...
};
```

此外，每一个“只一行代码”的 `Accept()` 函数本体都需要 `DocElementVisitor` 的定义，每一个 *concrete visitor* 都必须含入它感兴趣的 *classes*。这一切导致了复杂的依存关系。

真正的麻烦出现在“为 `DocElements` 添加新派生类”时。但是等等，我们不是不打算对 `DocElement` 继承体系添加任何新元素吗？`Visitor` 最适用于“稳定的继承体系”：你会对这个继承体系添加操作，但不触动继承体系本身。然而请你回忆一下，`Visitor` 为你完成这一任务的代价是造成难以在 *visited hierarchy*（本例之 `DocElement` 继承体系）中添加派生类。

但生活是充满变化的，让我们面对现实吧。世界上没有所谓“稳定的继承体系”这种东西。有时你不得不为 `DocElement` 继承体系添加一个新 `class`。就本例而言，要想增加一个从 `DocElement` 派生出来的 `vectorGraphic` `class`，你必须完成以下事情：

- 进入 `DocElementVisitor.h` 文件，为 `vectorGraphic` 新增一个前置声明。
- 在 `DocElementVisitor` 中新增一个纯虚重载函数，像下面这样：

```
class DocElementVisitor
{
    ... as before ...
    virtual void VisitVectorGraphic(VectorGraphic&) = 0;
};
```
- 来到每一个 *concrete visitor* 中实作 `visitVectorGraphic()`。根据任务的不同，这个函数可以“不做任何事情”。或者你可以将 `ElementVisitor::VisitVectorGraphic()` 定义为一个“不做任何事情”的函数，而不是一个纯虚函数，但这种情况下，如果你没有在 *concrete visitors* 中实作它，编译器不会提醒你。
- 在 `VectorGraphic` `class` 中实作 `Accept()`。一定不要忘记这一点！如果你直接从 `DocElement` 派生出一个 `class`，但忘了实作 `Accept()`，没问题，编译器会提醒你。但如果你派生自另一个 `class`，例如 `Graphic`，而 `Graphic` 定义了 `Accept()`，这种情况下一旦你忘记“令 `vectorGraphic` 可被访问 (visitable)”，编译器不会提醒你半句话。这个“臭虫”只有到执行期才可能被发现——也就是当你注意到你的 `Visitor framework` 拒绝调用任何 `VisitVectorGraphic()` 时。这的确是个难以对付的“臭虫”。

结论：`DocElement` 继承体系和 `DocElementVisitor` 继承体系中的的一切都需要重新编译，你得机械式地增加大量代码，而这一切只是为了让事情得以运转。根据项目的大小，这一需求小则

令人生厌，大则让人完全无法接受。

Robert Martin (1996) 首创了一种有趣的“Visitor 模式变体”，它利用 `dynamic_cast` 来消除“循环性”。他的做法是，为 *visitor hierarchy* 定义一个 *strawman*（稻草人）base class，例如 `BaseVisitor`。它只是型别信息的载体，不具任何内容，因此完全不具耦合性。*visited hierarchy* 的 `Accept()` 成员函数接受 *reference to BaseVisitor* 为参数，并对此参数施行 `dynamic_cast` 以找到相匹配的 *visitor*。如果 `Accept()` 找到了相匹配的 *visitor*，它会从 *visited hierarchy* 跳转至 *visitor hierarchy*。

这个解法乍听之下可能很怪异，实际上却非常简单。我们来看看，对于我们的这个 `DocElement/DocElementVisitor` 设计，如何实现一个 *acyclic visitor*（非循环访问者）。首先为 *visitor* 定义一个“稻草人”基类（*strawman base class*）：

```
class DocElementVisitor
{
public:
    virtual void ~DocElementVisitor() {}
};
```

“什么也没做”的虚析构函数实际上做了两件重要的事情。首先它为 `DocElementVisitor` 提供了 RTTI（执行期型别信息）能力（译注：这是虚函数带来的特性）。第二，这个虚析构函数确保 `DocElementVisitor` 对象具有正确的多态析构行为。如果一个多态继承体系没有虚析构函数，那么当你通过 *pointer to base object* 摧毁 *derived object* 时，将造成不确定的行为。因此，通过这一行代码，我们完美地解决了两个危险问题。

`DocElement` class 的定义维持不变。我们关心的是它对纯虚函数 `Accept(Doc-ElementVisitor&)` 的定义。

现在，针对 *visited hierarchy*（以 `DocElement` 起源的那个）中的每一个 *derived class*，我们定义一个小型的 *visiting class*，其中只有一个函数，名为 `VisitXxxx()`。例如，针对 `Paragraph`，我们定义如下：

```
class ParagraphVisitor
{
public:
    virtual void VisitParagraph(Paragraph&) = 0;
};
```

`Paragraph::Accept()` 大致实作如下：

```
void Paragraph::Accept(DocElementVisitor& v)
{
    if (ParagraphVisitor* p =
        dynamic_cast<ParagraphVisitor*>(&v))
    {
        p->VisitParagraph(*this);
    }
}
```

```

else
{
    optionally call a catch-all function
}
}

```

这里的主角是 `dynamic_cast`，有了它，执行期系统 (runtime system) 就能够判断 `v` 是否实际上是一个“也实现了 `ParagraphVisitor`”的对象的子对象；如果是，我们会得到一个“指向那个 `ParagraphVisitor`”的指针。

`RasterBitmap` 和其他 `DocElement` derived classes 也对 `Accept()` 做了类似定义。最后，一个具体的 visitor 诞生了，派生自“`DocElementVisitor`，及它所在意之所有 classes 的原型 visitors”。例如：

```

class DocStats :
    public DocElementVisitor, // Required
    public ParagraphVisitor, // Wants to visit Paragraph objects
    public RasterBitmapVisitor // Wants to visit RasterBitmap
                                // objects
{
public:
    void VisitParagraph(Paragraph& par)
    {
        chars_ += par.NumChars();
        words_ += par.NumWords();
    }
    void VisitRasterBitmap(RasterBitmap&)
    {
        ++images_;
    }
};

```

形成的 class 结构如图 10.2 所示。图中水平虚线表示相互之间有依存层 (dependency layers)，垂直线表示继承体系之间的绝缘性 (insulation)。请注意 `dynamic_cast`，看看它如何利用 strawman class `DocElementVisitor` 为你提供“从一个继承体系跳到另一个继承体系”的魔术。

上述 classes 结构牵涉大量细节和互动，但是基本结构很简单。让我们来分析一个事件流程 (event flow)，藉此做个摘要总结。假设有一个指针 `pDocElem`，指向一个 `DocElement`，但这个 `DocElement` 的动态 (实际) 型别是 `Paragraph`。然后执行以下动作：

```

DocStats stats;
pDocElem->Accept(stats);

```

那么以下事件会依次发生：

1. `stats` 对象被自动转换为 reference to `DocElementVisitor`，因为 `DocStats` 公开 (publicly) 继承自 `DocElementVisitor`。
2. 虚函数 `Paragraph::Accept()` 被调用。
3. `Paragraph::Accept()` 对“接收到的 `DocElementVisitor` 对象”的地址施行 `dynamic_cast`

<ParagraphVisitor*>。由于这个对象的动态型别是 `DocStats`，并且因为 `DocStats` 公开继承自 `DocElementVisitor` 和 `ParagraphVisitor`，所以转型会成功。这里也就是远程传输（teleporting）的发生地点！

4. 现在，`Paragraph::Accept()` 得到了一个指针，指向 `DocStats` 对象中的 `ParagraphVisitor` 成分。`Paragraph::Accept()` 会根据这个指针调用虚函数 `VisitParagraph()`。
5. 虚调用到达 `DocStats::VisitParagraph()`。此函数接收一个 *reference to visited paragraph* 作为参数，访问（visit）完成。

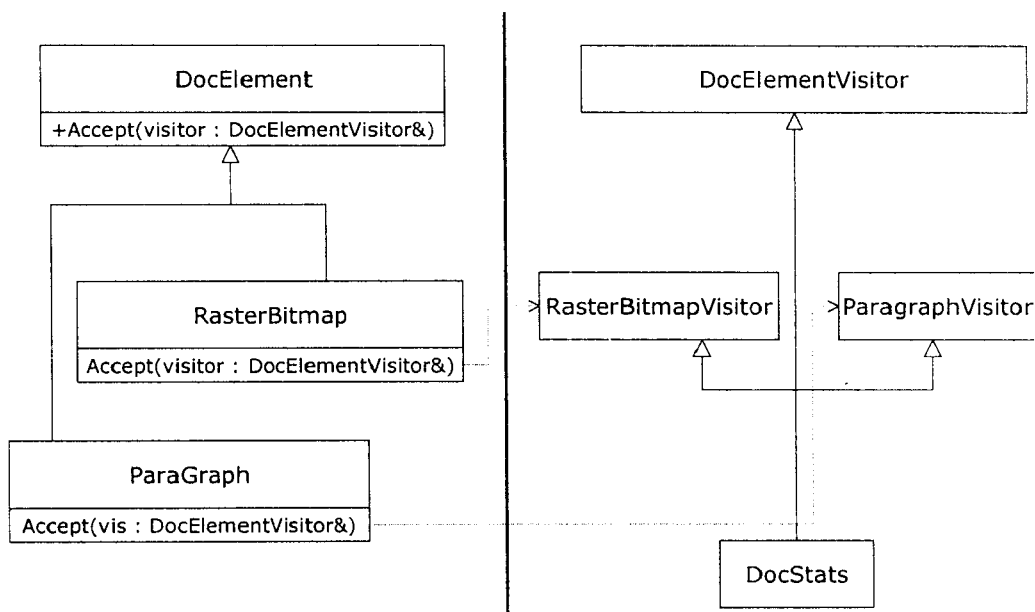


图 10.2 Acyclic Visitor 的 class 结构

我们来分析一下新的依存关系。

- `DocElement` class 定义式在名称上依存 `DocElementVisitor`。“名称上依存”意味着：只要前置声明 `DocElementVisitor`，问题就可以解决。
- `ParagraphVisitor` ——或更广泛地说是所有 `xxxVisitor` base classes——名称上依存于它将访问 (visits) 的 classes。
- `Paragraph::Accept()` 的实作码完全依存于 `DocElementVisitor`。“完全依存”意味着：编译代码时需要 class 的完整定义。
- 任何 *concrete visitor class* 的定义完全依存于 `DocElementVisitor`，以及它要访问的所有名为 `xxxVisitor` 的 base visitors。

Acyclic Visitor 模式消除了循环依存性，但同时它也给你留下了更多需要完成的工作。一般来说，从现在开始，你得平行维护两组 classes：源自 `DocElement` 的 *visited hierarchy*，以及 *visitor classes* `xxxVisitor` ——每一个 *concrete visited class* 都对应一个 `xxxVisitor`。同时维护两个 class 继承体系并不是件令人向往的好事，因为那需要许多素养和专注力。

请注意一般的 Visitor 和 Acyclic Visitor 效能上的对比：后一种模式会造成使用过程中一个额外的动态转型。其时间成本可能是常数，也可能根据程序中 *polymorphic classes* (多态类) 的个数，呈指数增长或线性增长——具体取决于你的编译器。如果效能对你的程序来说很重要，这一成本就会显得很突出，所以某些场合下，你会被迫使用一般的 Visitor，并继续面对循环依存性。

由于存在这种无情的现实，Visitor 成为一种争议性模式也就不足为奇了。即使 GoF 中的一员 Ralph Gamma 也曾说过：在他列出的十个排在后面的模式中，Visitor 排在最后 (Vlissides 1999)。

就算 Visitor 笨拙、呆板、难以扩充、难以维护，但只要有坚定的信念和勤奋的精神，我们照样可以组装出 Visitor 的程序库实作品，并做到双赢：易于使用、扩充和维护。下一节演示实作技法。

10.4 Visitor 之泛型实作

让我们将实作划分为两个主要单元：

- *Visitable classes*。这些 classes 都隶属于我们“想访问 (想为其添加操作)”的那个继承体系。
- *Visitor classes*。这些 classes 定义并实作出实际操作。

表 10.1 组件名称

名称	隶属于	用来表示
BaseVisitable	程序库	可受访之所有继承体系的 root class
Visitable	程序库	一个 mix-in class, 为可受访 (visitable) 继承体系中的 class 提供“可受访”特性
DocElement	应用程序	一个示例用的 class, 可受访 (visitable) 继承体系的 root class
Paragraph, RasterBitmap	应用程序	两个示例用的可受访具体类 (visitable concrete class), 从 DocElement 派生而来
Accept	程序库和应用程序	成员函数, 被调用用以访问可受访 (visitable) 继承体系
BaseVisitor	程序库	visitor 继承体系的 root class
visitor	程序库	一个 mix-in class, 为 visitor 继承体系中的 class 提供 visitor 特性
Statistics	应用程序	一个示例用的 visitor concrete class
Visit	程序库和应用程序	成员函数, 被可受访 (visitable) 元素针对其 visitors 回调

我们的做法简单一致：设法将代码尽可能抽取到程序库中。如果成功做到了这一点，我们得到的将是“大大简化了的依存关系”。也就是说，两大块 (visitor 和 visitable) 之间并不相互依存，而是都依存于程序库。这很不错，因为较之于应用程序，程序库理当更稳定，更可被信赖，更可被依存。

首先，我们设法实作一个泛型的 Acyclic Visitor，因为在处理依存性和分离性方面，它有较好的行为。然后，为了效率，我们会对它进行调整。最后再回过头来实作一个典型的 GoF Visitor，这个 Visitor 比较快速，但牺牲了一些灵活性。

在 Visitor 实作讨论中，我们采用表 10.1 列出的名称。表中一些名称实际描述的是 class templates，或者更确切地说，在代码逐渐泛化的过程中，它们将“变成”class templates。眼下最重要的是，对于这些名称所代表的物体 (entities)，我们得进行定义。

首先我们把注意力放在 visitor 继承体系上。事情很简单，我们必须为用户提供一些 base classes，它们也就是“针对某型别定义了 visit() 操作函数”的那些 classes。此外，我们还必须提供“strawman” class；根据 Acyclic Visitor 模式的需求，动态转型 (dynamic cast) 会用到 strawman class。代码如下：

```
class BaseVisitor
{
public:
    virtual ~BaseVisitor() {}
};
```

这个 class 没有多大复用性，但得有人写出这个小小的 class。

现在，我们提供一个简单的 Visitor class template。Visitor<T> 定义了一个纯虚函数，用以访问型别为 T 的对象。

```
template <class T>
class Visitor
{
public:
    virtual void Visit(T&) = 0;
};
```

一般情况下 Visitor<T>::Visit() 会传回某物，而不是传回 void。Visitor<T>::Visit() 可以经由 Visitable::Accept() 传递有用的结果。所以，我们为 Visitor 增加第二个 template 参数：

```
template <class T, typename R = void>
class Visitor
{
public:
    typedef R ReturnType;
    virtual ReturnType Visit(T&) = 0;
};
```

无论何时，如果想访问一个继承体系，首先请从 BaseVisitor 派生出你的 ConcreteVisitor；你想访问多少个型别，就得有多少个 visitor 具现体。

```
class SomeVisitor :
    public BaseVisitor // required
    public Visitor<RasterBitmap>,
    public Visitor<Paragraph>
{
public:
    void Visit(RasterBitmap&); // visit a RasterBitmap
    void Visit(Paragraph &); // visit a Paragraph
};
```

代码看上去干净、简单、易于使用。采用这一结构，你的代码访问哪些 classes 就变得一清二楚。更棒的是，如果你没有定义所有 Visit 函数，编译器就不允许你具现 SomeVisitor。在名称方面，SomeVisitor class 的定义和它所访问的 classes (RasterBitmap 和 Paragraph) 的名称之间有依存性，这很容易让人理解，因为 SomeVisitor “知道” 这些型别，而且 “需要” 对这些型别执行特定操作。

现在，我们已经完成了 visitor 这一端的实作。我们定义了一个简单的 base class，用来为

`dynamic_cast` (`BaseVisitor`) 和 `visitor class template` 提供帮助, 后者为每一种 *visited type* 生成一个纯虚函数。

至此, 我们写的代码虽然不多, 但颇具复用性。`visitor class template` 为每一个 *visited class* 各自生成一个型别。这样一来, 程序库客户就不需在整份实作内四处乱扔这些小型 *classes* (例如 `ParagraphVisitor` 和 `RasterBitmapVisitor`)。`visitor class template` 生成的型别将确保与 *visitable classes* 之间的联系。

这就将我们带到 *visitable* (可受访) 继承体系面前。如前一节所述, 在 *Acyclic Visitor* 实作之中, *visitable* 继承体系担负以下职责:

- 声明一个纯虚函数 `Accept()`, 其参数为 *base class* 中的一个 *reference to Visitor*。
- 在每一个 *derived class* 中, 改写并实作那个 `Accept()` 函数。

为履行第一条职责, 我们为 `BaseVisitable` *class* 增加一个纯虚函数, 并要求 `DocElement`——广而言之泛指程序范围中任何一个 *visitable* 继承体系的 *root class*——从 `BaseVisitable` *class* 继承。`BaseVisitable` 大致像这样:

```
template <typename R = void>
class BaseVisitable
{
public:
    typedef R ReturnType;
    virtual ~BaseVisitable() {}
    virtual ReturnType Accept(BaseVisitor&) = 0;
};
```

真简单! 当我们履行第二个职责时, 也就是在程序库 (而非客户代码) 中实作 `Accept()` 时, 事情变得有趣起来。实作 `Accept()` 函数的工作量并不大, 但每一个客户 *class* 都得处理它, 这很让人讨厌。如果它们属于程序库的一部分就好了。仔细听好, 就像 *Acyclic Visitor* 模式要求的那样, 如果 *T* 是个 *visited class*, 那么 `T::Accept()` 实作码会将 `dynamic_cast<Visitor<T>*>` 施行于 `BaseVisitor` 身上。如果转换成功, `T::Accept()` 会跳转至 `Visitor<T>::Visit()`。但是正如 10.2 节所说, 将 `Accept()` 提取至 `BaseVisitable` *class* 是不行的, 因为在其中 **this* 的静态型别无法为 *visitor* 提供适当的型别信息。

我们需要一种“能够将 `Accept()` 实作于程序库中, 并将 `Accept()` 注入应用程序之 `DocElement` 继承体系内”的方法。可惜 C++ 没有提供这种直接机制。使用虚拟继承 (*virtual inheritance*) 是一种间接法, 但此法称不上一流, 而且存在不可忽视的成本。我们得求助于宏 (*macro*), 并要求这个 *visitable* 继承体系中的每一个 *class* 都要在其 *class* 定义式中使用这个宏。

宏的使用必然伴随着它们所带来的笨拙; 因此, 宏的使用并不是一个容易做出的决定。但是采用其他各种方案都不能增加多少好处, 反而会消耗大量时间和空间。大家都知道, C++ 程序员是注重实际的人, 效率因素足以让他们时常依靠“宏”而非“深奥但缺乏效率的技巧”。

定义宏时,唯一最重要的原则是:我们要让宏本身尽量少做事,要尽快将它转入“真正”的物体 (function, class 等等)。我们可以像下面这样为 *visitable classes* 定义一个宏:

```
#define DEFINE_VISITABLE() \
    virtual ReturnTyp Accept(BaseVisitor& guest) \
    { return AcceptImpl(*this, guest); }
```

客户必须将 `DEFINE_VISITABLE()` 插入 *visitable* 继承体系内的每一个 class 内。在上述定义式中, `Accept()` 成员函数被转至另一个函数: `AcceptImpl()`, 那是一个 *template* 函数, 根据 **this* 的型别而参数化。这样, `AcceptImpl()` 就得到了它极需要的静态型别。我们在继承体系的最顶层 *base class* 中定义 `AcceptImpl()`, 这么一来所有派生类都能取用它。以下是修改后的 *BaseVisitable class*:

```
template <typename R = void>
class BaseVisitable
{
public:
    typedef R ReturnTyp;
    virtual ~BaseVisitable() {}
    virtual ReturnTyp Accept(BaseVisitor&) = 0;
protected: // Give access only to the hierarchy
    template <class T>
    static ReturnTyp AcceptImpl(T& visited, BaseVisitor& guest)
    {
        // Apply the Acyclic Visitor
        if (Visitor<T>* p = dynamic_cast<Visitor<T>*>(&guest))
        {
            return p->Visit(visited);
        }
        return ReturnTyp();
    }
};
```

现在, 我们设法将 `AcceptImpl` 转移进入程序库, 这一点很重要。其意义不仅仅在于使客户的工作获得了自动化, 而且, 由于 `AcceptImpl()` 出现在程序库中, 我们有机会根据各种不同的设计条件去调整其实作; 关于这一点, 稍后会更清楚。

Visitor/Visitable 的最终设计对用户隐藏了大量细节, 而且使用上没有繁文缛节, 工作起来也十分有效。以下是到目前为止, 我们这个泛型 *Acyclic Visitor* 实作品的基本架构。

```
// Visitor part
class BaseVisitor
{
public:
    virtual ~BaseVisitor() {}
};
template <class T, typename R = void>
class Visitor
{
public:
```

```

typedef R ReturnType; // Available for clients
virtual ReturnType Visit(T&) = 0;
};

// Visitable part
template <typename R = void>
class BaseVisitable
{
public:
    typedef R ReturnType;
    virtual ~BaseVisitable() {}
    virtual R Accept(BaseVisitor&) = 0;
protected:
    template <class T>
    static ReturnType AcceptImpl(T& visited, BaseVisitor& guest)
    {
        // Apply the Acyclic Visitor
        if (Visitor<T>* p = dynamic_cast<Visitor<T>*>(&guest))
        {
            return p->Visit(visited);
        }
        return ReturnType();
    }
};

#define DEFINE_VISITABLE() \
    virtual ReturnType Accept(BaseVisitor& guest) \
    { return AcceptImpl(*this, guest); }

```

准备测试了吗？以下便是：

```

class DocElement : public BaseVisitable<>
{
public:
    DEFINE_VISITABLE()
};

class Paragraph : public DocElement
{
public:
    DEFINE_VISITABLE()
};

class MyConcreteVisitor :
    public BaseVisitor, // required
    public Visitor<DocElement>, // visits DocElements
    public Visitor<Paragraph> // visits Paragraphs
{
public:
    void Visit(DocElement&) { std::cout << "Visit(DocElement&) \n"; }
    void Visit(Paragraph&) { std::cout << "Visit(Paragraph&) \n"; }
};

```

```
int main()
{
    MyConcreteVisitor visitor;
    Paragraph par;
    DocElement* d = &par; // "hide" the static type of 'par'
    d->Accept(visitor);
}
```

这个小程序会输出:

```
Visit(Paragraph&)
```

意味着一切工作正常。

当然, 在这个玩具小程序中, 我们组装出的东西并未完全展现其强大威力。然而经历过前一节白手起家实现 visitors 的痛苦历程后, 现在的我们确实实拥有了一个工具。要想正确产生 *visitable* 继承体系并访问之, 一切都变得更容易。

让我们回顾一下, 定义 *visitable* 继承体系时, 我们必须执行哪些动作。

- 从 *BaseVisitable<YourReturnType>* 派生出你的继承体系的 root class。
- 在 *visitable* 继承体系的每一个 *SomeClass* class 中添加 *DEFINE_VISITABLE()* 宏。(至此, 你的继承体系可以被访问了, 但是看不到对任何 Visitor 的依赖!)
- 从 *BaseVisitor* 派生出每一个 *concrete visitor* *SomeVisitor*。此外, 针对每一个你需要访问的 class *X*, 从 *Visitor<X, YourReturnType>* 派生出 *SomeVisitor*。为每一个 *visited type* (受访型别) 改写成员函数 *Visit()*。

形成的依赖关系图很简单。*SomeVisitor* class 定义式“名称上依存”于每一个 *visited class*。成员函数 *Visit()* 的实作“完全依存”于被操作的 classes。

大致就是这样。和前面讨论的 Visitor 实作相比, 这份实作为我们带来了更多方便。在它的帮助下, 如今我们可以经由一种井然有序的方式来建造 *visitable* 继承体系; 同时, 我们还将客户代码的数量和依存性降至最低程度。

某些特殊情况下你可能希望直接实作 *Accept()*, 而不是使用 *DEFINE_VISITABLE* 宏。假设你定义了一个派生自 *DocElement* 的 *Section* class; 一个 *Section* 包含若干个 *Paragraphs*。对于一个 *Section* 对象, 你可能想访问其中每一个 *Paragraph*。这种情况下你可能会手工实作 *Accept()*:

```
class Section : public DocElement
// Won't use the DEFINE_VISITABLE() macro
// because it's going to implement Accept by itself
{
    ...
    virtual ReturnType Accept(BaseVisitor& v)
    {
        for (each paragraph in this section)
        {
            current_paragraph->Accept(v);
        }
    }
}
```



```

    }
}
};

```

如你所见，使用这份 Visitor 实作品时，没有什么事情不能做。这份代码只是为了将你从繁重的工作中解放出来——如果白手起家，你就不得不做这些工作。

我们已经完成了对 Visitor 核心实作的定义，其中包括几乎所有“和基本访问操作密切相关的东西”。请继续阅读，还有很多内容等着你。

10.5 再论 "Cyclic" Visitor

大多数情况下，前一节定义的泛型 Acyclic Visitor 应该能够令人满意。但是，如果你开发的是一个对速度要求很高的应用程序，Accept() 中的动态转型可能会令你忧心忡忡，你的速度测试结果也会令你沮丧。原因是：如果对某一对象应用 dynamic_cast，执行期支持系统将有很多事情要做。RTTI 代码必须判断“至目标型别的转换”是否合法；如果合法，就必须产生一个“指向目标型别的指针”。

让我们更详细地看看编译器设计者是如何完成这些事情的。一个合理的方案是，为程序中的每一个型别都分配一个唯一的整数标识符。要知道，即使是在异常处理时刻，整数标识符也是唾手可得，所以这是个十分明智的综合方案。接下来，在每个 class 的虚函数表格（virtual table）中，编译器添加一个表格（指针），表格中存储着这个 class 的所有子型别（subtype）的标识符。连同这些符，编译器必须存储这些子对象在这个大对象中的相对位置偏移量。这些便是正确执行动态转换时所需的足够信息。执行 dynamic_cast<T2*>(p1) 时，如果 p1 的型别是 pointer to T1，执行期支持系统会为 p1 遍历整个型别表格。如果找到了一个匹配的 T2，则执行期支持系统便执行必要的指针算术运算，并将结果传回。如果没有找到匹配型别，则传回一个 null 指针。一旦涉及细节（例如多重继承），动态转型的做法会更复杂、更缓慢。

基于一个 class 所具有的 base classes 的数目，上面描述的方案呈 $O(n)$ 复杂度。换句话说，随着你所使用的继承结构的深度和广度的递增，动态转型消耗的时间也呈线性增长。

另一个方案是使用 hash tables，可提高速度，但却得在内存用量上付出代价。还有一个方案是在整个程序中使用一个大型矩阵，从而使动态转型消耗常数时间，但会大大提高空间消耗——特别是在 classes 为数众多的程序中。（一种可能的做法是压缩这个矩阵；这一切都展示了 C++ 编译器设计者五彩缤纷的生活©）

结论是，dynamic_cast 一定得付出成本，那是无法预知的；对于某一程序的某一特定需要，这个成本还可能令人无法接受。这意味着我们应当扩充我们的 Visitor 实作品，以容纳 GoF 的 cyclic Visitor——它并没有使用 dynamic_cast，因此较为快速，但较难维护。

请回到本章起始，看看 GoF Visitor 是如何工作的。以下两点总结了 GoF Visitor 和 Loki 实作之 Acyclic Visitor 之间的主要区别。

- `BaseVisitor` class 不再是“稻草人”：它为每一个 *visited type* (受访型别) 都定义了一个 `visit()` 纯虚成员函数 (假设我们决定使用重载技术)。
- `AcceptImpl` 函数必须修改。宏 `DEFINE_VISITABLE()` 保持不变，非常好。

现在，简而言之，我们有个集合 (collection)，其中有我们想访问的型别。假设这些型别是 `DocElement`、`Paragraph` 和 `RasterBitmap`。如何表示并操纵一个型别集合呢？请看第 3 章，那里详细描述了 `typelists`，并定义了一个完整的 `typelist` 设施。

`Typelists` 正是这里所需要的。我们想将一个 `typelist` 作为 `template` 引数传递给 `CyclicVisitor` `template`，用以宣称“我希望这个 `CyclicVisitor` 能够访问这些型别”。这句话可以像下面这般优雅地说出：

```
// Forward declarations needed by the typelist below
class DocElement;
class Paragraph;
class RasterBitmap;

// Visits DocElement, Paragraph, and RasterBitmap
typedef CyclicVisitor
<
    void, // return type
    TYPelist_3(DocElement, Paragraph, RasterBitmap)
>
MyVisitor;
```

`CyclicVisitor` “名称上依存”于 `DocElement`、`Paragraph` 和 `RasterBitmap`。

让我们看看，对目前的架构，我们还需要添些什么。和第 9 章定义泛型 `Abstract Factory` 时一样，我们将采用相同步骤：

```
// Consult Visitor.h for the definition of Private::visitorBinder<R>
template <typename R, class TList>
class CyclicVisitor : public GenScatterHierarchy<TList,
    Private::VisitorBinder<R>::Result>
{
    typedef R ReturnType;
    template <class Visited>
    ReturnType Visit(Visited& host) {
        Visitor<Visited>& subObj = *this;
        return subObj.Visit(host);
    }
};
```

值得注意的是，`CyclicVisitor` 将 `visitor` 作为一个构件 (building block) 来使用。请参阅第 3 章关于这一技术的介绍，以及第 9 章一个类似的例子。本质上，针对 `TList` 这个 `typelist` 中的每一个型别 `T`，`CyclicVisitor` 都继承了一个 `visitor<T>`。

实际效果是，如果你将一个 `typelist` 传递给 `CyclicVisitor`，后者就会继承“经由 `typelist` 中的每一个型别具现出来的 `visitor`”，进而造成每个型别都声明了一个纯虚函数 `visit()`。换句

话说，功能上它就相当于“满足 GoF Visitor 模式要求”的一个 visitor。

藉由 typedef 指定你所选择的 CyclicVisitor (例如 MyVisitor) 后，需要做的只不过是你在你的 *visitable classes* 中恰当使用宏 `DEFINE_CYCLIC_VISITABLE(MyVisitor)`。例如：

```
typedef CyclicVisitor
<
    void, // return type
    TYPELIST_3(DocElement, Paragraph, RasterBitmap)
>
MyVisitor;

class DocElement
{
public:
    virtual void Accept(MyVisitor&) = 0;
};

class Paragraph : public DocElement
{
public:
    DEFINE_CYCLIC_VISITABLE(MyVisitor);
};
```

就这样！就像手工实作的 GoF Visitor，你还是得遵守一些规定。不同的是，如今你必须记住并自己动手的东西大大地减少了。如果有个继承体系，你想让我们的 GoF Visitor 能够访问它，只需完成以下工作：

- 前置声明 (forward declare) 这个继承体系中的所有 classes。
- 为 CyclicVisitor 写一个 typedef，以“返回型别”和“你想访问的型别清单”具现化这个 CyclicVisitor (姑且名为 MyVisitor)。
- 为你的 base class 定义一个纯虚函数 `Accept()`。
- 在你的继承体系的每一个 class 中添加 `DEFINE_CYCLIC_VISITABLE(MyVisitor)`，或者，在你打算提供不同行为的 classes 中手工实作 `Accept()`。
- 让你的每一个 *concrete visitors* 继承自 MyVisitor。
- 只要向 visitable 继承体系添加一个新 class，就得更新 MyVisitor template 具现体 (typedef)，然后…哎…重新编译。

和手工编写的 GoF Visitor 实作相比，这个泛型方案更加清晰。需要维护的地方大大减少，几乎只剩在你的 MyVisitor 型别定义中。

设计 Visitor 时，一条实用建议是：最好从 Acyclic Visitor 着手，因为它比较容易维护；只有在真正必须实施优化时，才转用 GoF Visitor。好处是，泛型组件很容易带给你体验——你只需修改一个声明，不必增加任何代码；Visitor 实作细节位于程序库中。只需调整客户和程序库之间的声明链 (declarative link)，你就可以获得两份不同的 Visitor 实作品。

10.6 变化手段

Visitor 有大量的变化形式和定制形式。本节就这方面的一些非常手段作了专门的介绍；你也许在某些场合下需要实作出这些手段。

10.6.1 Catch-All 函数

我们在 10.2 节讨论了 catch-all 函数。Visitor 可能会碰上一个从 base class（前例中的 DocElement）派生而来的未知型别。在这种情况下，你如果不想得到一个编译期错误，就得以执行期执行某一缺省行为。

针对我们的泛型组件所提供的 GoF Visitor 实作品和 Acyclic Visitor 实作品，我们来分析一下 catch-all 问题。

对 GoF Visitor 而言，情况非常简单。在传给 CyclicVisitor 的 typelist 中，如果你包含了“你的继承体系的 root class”，就是给了自己一个“实现 catch-all 函数”的机会。否则你就是选择了“编译期错误”这一条路。以下代码示范这两种选择：

```
// Forward declarations needed for the GoF Visitor
class DocElement; // Root class
class Paragraph;
class RasterBitmap;
class VectorizedDrawing;

typedef CyclicVisitor
<
    void, // Return type
    TYPELIST_3(Paragraph, RasterBitmap, VectorizedDrawing)
>
StrictVisitor; // No catch-all operation;
                // will issue a compile-time error if you try
                // visiting an unknown type

typedef CyclicVisitor
<
    void, // return type
    TYPELIST_4(DocElement, Paragraph, RasterBitmap,
                VectorizedDrawing)
>
NonStrictVisitor; // Declares Visit(DocElement&), which will be
                  // called whenever you try visiting
                  // an unknown type
```

这一切都非常简单。现在，让我们来谈谈 Acyclic Visitor 泛型实作中的 catch-all 函数。

在 Acyclic Visitor 中，一件有趣的事情发生了。本质上 catch-all 涉及的是“一个已知的 visitor 访问一个未知的 class”；但此处出现的问题却相反：一个未知的 visitor 访问一个已知的 class！

让我们再次看看 `Acyclic Visitor` 如何实作 `AcceptImpl()`。

```
template <typename R = void >
class BaseVisitable
{
    ... as above ...
    template <class T>
    static ReturnTpe AcceptImpl(T& visited,
        BaseVisitor& guest)
    {
        if (Visitor<T>* p =
            dynamic_cast<Visitor<T>*>(&guest))
        {
            return p->Visit(visited);
        }
        return ReturnTpe(); // Attention here!
    }
};
```

假设你在你的 `DocElement` 继承体系中增加一个 `VectorizedDrawing`，而且是经由一种很隐蔽的方式——没有通知任何 *concrete visitors*。那么，只要访问 `VectorizedDrawing`，动态转型至 `Visitor<VectorizedDrawing>` 的行动就会失败。这是因为你现有的 *concrete visitors* 不知道这个新型别，因而未能从 `Visitor<VectorizedDrawing>` 派生。由于动态转型失败，程序选择了另一条路线，传回 `ReturnTpe` 的缺省值。这正是 `catch-all` 函数可以发挥作用的地方。

在我们的 `AcceptImpl` 函数中，`return ReturnTpe()` 这个动作被写死，这就造成了非常苛刻的设计限制，没有留下变化余地。为此，我们可以提供一个 `policy`，用来执行 `catch-all` 行为：

```
template
<
    typename R = void,
    template <typename, class> class CatchAll = DefaultCatchAll
>
class BaseVisitable
{
    ... as above ...
    template <class T>
    static ReturnTpe AcceptImpl(T& visited,
        BaseVisitor& guest)
    {
        if (Visitor<T>* p =
            dynamic_cast<Visitor<T>*>(&guest))
        {
            return p->Visit(visited);
        }
        // Changed
        return CatchAll<R, T>::OnUnknownVisitor(visited, guest);
    }
};
```

无论这个设计有什么样的要求, **CatchAll** policy 都能实现。它可以传回一个缺省值或错误码、抛出异常、为“受访对象”调用一个虚成员函数、多次尝试动态转型... **OnUnknownVisitor** 的实作很大程度取决于具体的设计需求。有些情况下, 你可能想访问这个继承体系中的所有型别, 另一些情况下你可能想随意访问某些型别, 并默默地忽略其他所有型别。Acyclic Visitor 实作品采用的是第二种态度: 所以, 缺省的 **CatchAll** 大致像这样:

```
template <class R, class Visited>
struct DefaultCatchAll
{
    static R OnUnknownVisitor(Visited&, BaseVisitor&)
    {
        // Here's the action that will be taken when
        // there is a mismatch between a Visitor and a Visited.
        // The stock behavior is to return a default value
        return R ();
    }
};
```

如果需要不同的行为, 你只需在 **BaseVisitable** 中采用另一个不同的 **template** 即可。

10.6.2 非严格访问 (NonStrict Visitation)

或许是人的天性, “鱼与熊掌得兼”的理想是每一个程序员心中的信念。如果可能, 程序员都希望有一个快速、非耦合、灵活的 visitor, 这个 visitor 会解读他们的心思, 判断某个疏忽究竟是愚蠢的错误还是有意的决定。另一方面说, 与其客户不同^③, 程序员是很实际、很朴实的人, 你可以和他们就利弊优劣讨价还价。

按这一思路进行下去, 可以想见, **CatchAll** 的灵活性一定会让 GoF Visitor 的用户羡慕不已。事实上 GoF Visitor 实作品很严谨周密, 它为每一个“受访型别”都声明一个纯虚函数。你必须从 **BaseVisitor** 派生, 并实作每一个 **Visit()** 重载函数; 如果不这样, 代码就无法编译。

但是有时你并不真的想访问清单中的每一个型别。你不希望这个 framework 如此严格。你希望有所选择: 要不你就为每个型别实作 **Visit**, 否则 **OnUnknownVisitor** 会因为你完成了 **CatchAll** 而被自动调用。

对于这种情况, Loki 提供了一个 **BaseVisitorImpl** class。它继承自 **BaseVisitor** 并运用 **typelist** 技术以接纳 **typelists**。你可以在 Loki 的 **Visitor.h** 中找到它。

10.7 摘要

本章探讨了 Visitor 模式及其相关问题。一般来说, Visitor 可让你为一个 class 继承体系添加虚函数, 而无需修改那个继承体系中的 classes。在某些场合, Visitor 带来灵巧、可扩充的设计。

但是 Visitor 也存在不少问题, 除非将它运用于极稳定的 class 继承体系, 否则, 使用它会存在很大的困难。Acyclic Visitor 有所改善, 但付出的代价是效率。

经由认真的设计和高级实作技术的运用，泛型 Visitor 组件完全可以发挥 Visitor 模式的最大作用。这份实作品保留了 Visitor 的全部功能，同时消除了它的大部分缺陷。

在实际应用中，Acyclic Visitor 应该是各种变化形式的首选，除非你想获得最大速度。如果速度很重要，你可以使用 GoF Visitor 之泛型实作品，它保留了“维护量少”（只有一个明显地方需要维护）、“编译成本合理”的特点。

这份泛型实作运用了高阶 C++ 编程技术，譬如动态转型、`typelists` 和 `partial specialization`（偏特化）。其成果是：无论针对何种具体需要，实作 visitor 时所需的一般性、重复性的工作都被提取到程序库中，由程序库为你完成。

10.8 Visitor 泛型组件要点概览

- 若要实作 Acyclic Visitor，请使用 `BaseVisitor`（作为 "strawman" base class）、`Visitor` 和 `Visitable`：

```
class BaseVisitor;

template <class T, typename R = void>
class Visitor;

template
<
    typename R = void,
    template<class, class> CatchAll = DefaultCatchAll
>
class BaseVisitable;
```

- `Visitor` 和 `BaseVisitable` 的第一个 `template` 参数分别是成员函数 `Visit()` 和 `Accept()` 的返回型别。它们被缺省为 `void`（这是 GoF 著作中的示例程序及大多数 Visitor 论述中采用的返回型别）。
- `BaseVisitable` 的第二个 `template` 参数是个 policy，用来处理 catch-all 问题（见 10.2 节）。
- 从 `BaseVisitable` 派生出“你的继承体系的 root class”，然后在这个继承体系的每一个 class 中使用宏 `DEFINE_VISITABLE()`，或手工实作出 `Accept(BaseVisitor&)`。
- 从 `BaseVisitor` 和 `Visitor<T>` 派生出你的 *concrete visitor classes*，其中 `T` 是你想访问的每一个型别。（译注：换言之这里使用了多重继承，multiple inheritance）
- 如果你要的是 GoF Visitor，请使用 `CyclicVisitor`：

```
template <typename R, class TList>
class CyclicVisitor;
```
- 在上述的 `TList` `template` 引数中指定受访型别（*visited types*）。
- 就像使用典型的 GoF Visitor 一样地使用 `CyclicVisitor`。

- 如果需要实现的只是 GoF Visitor 的一部分（亦即非严格变体），那么请从 `BaseVisitorImpl` 派生你的 `visitable` 继承体系。`BaseVisitorImpl` 实作出所有 `visit` 重载函数，用以调用 `OnUnknownVisitor()`。你可以改写其中的部分行为。
- `OnUnknownVisitor()` 是 `CatchAll` policy 的一个静态成员函数，提供了一个能够“为 Acyclic Visitor 捕捉所有情况”的场所。如果你使用 `BaseVisitorImpl()`，它提供的是一个能够“为 GoF Visitor 捕捉所有情况”的场所。Loki 提供的 `OnUnknownVisitor()` 实作品会传回一个根据你所选择的返回型别、被 default 构造函数构造完成的 `value`。你可以提供自定义的 `CatchAll` 实作品以改写这一行为。

11

Multimethods

本章以 C++ 为环境，定义、讨论和实现 multimethods。

C++ 的虚函数（virtual function）机制可以让你根据“一个”对象的动态型别来对调用动作进行分派（dispatch）。multi-methods 则可以让你根据“多个”对象的型别来对调用动作进行分派。通用性（处处皆可）实作需要语言本身的支持，是的，CLOS、ML、Haskell 和 Dylan 走的就是这条路。C++ 缺乏这样的支持，所以必须由程序库设计者模拟出来。

本章讨论了一些典型解法，以及每种解法之泛型实现。这些解法的特色是在速度、灵活性和依存性管理上的各种取舍。为了描述这种“根据多个对象对函数调用动作进行分派”的技术，本书使用（从 CLOS 借来的）*multimethod*（多重方法）和 *multiple dispatch*（多重分派）等术语。如果只是根据“两个”对象来进行 *multiple dispatch*，特别称为 *double dispatch*（双重分派）。

Multimethods 的实作是一个既让人痴迷又令人畏惧的工作，它窃走了设计者和程序员的大量甜美而有益的睡眠时间³⁶。

本章讨论的主题包括：

- 定义 multimethods。
- 识别哪些情况下需要“多物多态性”（multiobject polymorphism）。
- 讨论并实作三个各有利弊的 double dispatchers。
- 加强 double-dispatch 引擎。

阅读本章之后，你将牢固掌握 multimethods 的典型使用情况。此外，你将学会使用和扩充 Loki 提供的用以实现 multimethods 的数个坚固的泛型组件。

本章对 multimethods 的讨论只限于两个对象（亦即 double dispatch）。你可以使用相关技术，将支持的对象数目扩充至更多。大多数情况下你很可能都只是根据两个对象进行分派，这时候你可以直接使用 Loki。

³⁶ 如果你在实作 multimethods 的过程上有困难，我希望你可以将本章看做有助睡眠的东西——这并不是说它实际上有催眠作用 ☺。

11.1 什么是 Multimethods?

在 C++ 中, 多态 (polymorphism) 本质上意味着: 基于编译期 (compile-time) 或执行期 (runtime) 环境条件的不同, 某一函数调用可被绑定于不同的实作 (函数体) 上。

C++ 实现两种多态:

- 编译期多态 (compile-time polymorphism): 重载 (overloading) 和 template 函数支持之³⁷。
- 执行期多态 (runtime polymorphism): 由虚函数 (virtual functions) 实现。

重载是多态的简单形式, 它让多个具有相同名称的函数可以在同一个 scope 共存。只要函数具备不同的参数表, 编译器便能够在编译期区分它们。重载只不过提供了语法上的便利, 是一种简便的语法上的缩写形式。

template 函数是一种静态分派 (static dispatch) 机制。它提供的是更复杂的编译期多态。

在虚函数调用中, C++ 执行期系统 (而非编译器) 将确定实际调用哪一个函数。执行期间, 虚函数会将一个名称绑定于一个函数实体。至于究竟哪一个函数被绑定, 取决于参与“虚调用”的对象的动态型别 (dynamic type)。

现在, 我们看看, 如何将这三种多态机制扩展至多重对象。重载和 template 函数可以很自然地扩展到多重对象, 因此, 它们都接受多重参数, 而且复杂的编译规则则会负责函数的选择。

不幸的是, 虚函数——C++ 实现“执行期多态”的唯一机制——只适用于单一对象。即使从调用语法 `obj.Fun(arguments)` 来看, `obj` 的角色也凌驾于 `arguments` 之上。(其实你也可以将 `obj` 看做是 `Fun` 的另一个引数, 这个引数在 `Fun` 内部可通过 `*this` 取得。例如 Dylan 语言在处理 dot call 语法时, 就只是将它视为一般函数调用机制的特殊表达式来处理)

我们将 *multimethods* 或 *multiple dispatch* 定义为一种机制, 这一机制会根据函数调用所涉及的多对象的动态型别, 将调动作分派给适当的 (吻合的) 具体函数。由于编译期的“多物多态性” (multiobject polymorphism) 已由语言支持, 所以我们只需实现执行期的“多物多态性”。啊, 别急, 很多东西需要探讨。

11.2 何时需要 Multimethods?

如何判断是否需要 multimethods? 很简单。假设你有一个操作函数, 通过 pointer (or reference) to base classes 的形式来处理多个多态对象, 而你希望该操作函数的行为能够根据多个 (一个以上) 对象的动态型别的不同而变化, 这就是了。

³⁷ “多态”的一个更一般性观点中, 自动转型 (automatic conversions) 被视为一种最粗糙的多态形式。例如 `std::sin` 原本用于 `double` 型别, 有了自动转型, 它也可以经由 `int` 调用, 但这种通过强制手段产生的多态只是表面上的, 因为对于不同的两个型别, 调用的是同一个函数。

通过 `multimethods` 解决的典型问题之一就是“碰撞”问题。举个例子，假设你想编写一个视频游戏，其中可移动对象都派生自 `GameObject` abstract class。你希望这些对象的碰撞效果系根据“哪两个对象发生碰撞”而有所不同：可能的组合包括太空船碰撞行星、太空船碰撞太空站、太空站碰撞行星³⁸。

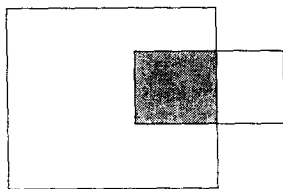


图 11.1 在两个形状的重叠处绘制影线

本章所用的例子如下。假设你想标示图形对象的重叠区。你编写了一个绘图程序，允许用户定义矩形、圆形、椭圆形、多边形和其他形状。程序的基本设计采用典型的面向对象思想：定义一个名为 `Shape` 的 abstract class，让所有具体形状都从它派生，然后将整个图形以一个“由 pointers to `Shape` 组成的集合 (collection)”来操纵。

现在假设客户要求一项功能：如果两个封闭形状相交，相交处以某种方式绘出，而且绘制方式应当不同于两个形状中的任何一个——例如可以在相交区域绘出影线（见图 11.1）。

找出一个唯一算法来为任何相交区绘制影线，是非常困难的事。在两个矩形的相交区绘制影线，和在椭圆形和矩形的相交区绘制影线，算法大不相同（而且复杂得多）。此外，太过一般性的算法——例如图素层次 (pixel level) 的算法——往往十分缺乏效率，因为有些相交情况（譬如矩形和矩形）其实很简单。

这里你需要的是“一组”算法：每个算法专门针对两种形状，例如矩形-矩形、矩形-多边形、多边形-多边形、椭圆-矩形、椭圆-多边形、椭圆-椭圆。执行期间当用户在屏幕上移动这些形状时，你选择并调用适当的算法，这些算法会执行快速运算，在重叠区域绘制影线。

由于你是通过 `pointer to Shape` 来操纵所有图形对象，所以，选择合适的算法时，你缺少必要的类型信息。你拥有的仅是 `pointer to Shape`。这里涉及两个对象，所以一般的（单纯的）虚函数无法解决问题。此时你得使用 `double dispatch`。

11.3 Double Switch-on-Type: 暴力法

实现 `double dispatch` 的最直接方式是所谓的 `double switch-on-type`。你可以尝试将第一个对象动态转型为每一个可能的左侧型别 (left-hand types)，并在每一个分支中针对第二个引数做同样

³⁸ 这个例子和名称均借自 Scott Meyers 的《*More Effective C++*》（1996a）条款 31。

的事。一旦确知两个对象的型别，你就知道该调用哪个函数了。代码看起来像这样：

```
// various intersection algorithms
void DoHatchArea1(Rectangle&, Rectangle&);
void DoHatchArea2(Rectangle&, Ellipse&);
void DoHatchArea3(Rectangle&, Poly&);
void DoHatchArea4(Ellipse&, Poly&);
void DoHatchArea5(Ellipse&, Ellipse&);
void DoHatchArea6(Poly&, Poly&);

void DoubleDispatch(Shape& lhs, Shape& rhs)
{
    if (Rectangle* p1 = dynamic_cast<Rectangle*>(&lhs))
    {
        if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
            DoHatchArea1(*p1, *p2);
        else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs))
            DoHatchArea2(*p1, *p2);
        else if (Poly* p2 = dynamic_cast<Poly*>(&rhs))
            DoHatchArea3(*p1, *p2);
        else
            Error("Undefined Intersection");
    }
    else if (Ellipse* p1 = dynamic_cast<Ellipse*>(&lhs))
    {
        if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
            DoHatchArea2(*p2, *p1);
        else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs))
            DoHatchArea5(*p1, *p2);
        else if (Poly* p2 = dynamic_cast<Poly*>(&rhs))
            DoHatchArea4(*p1, *p2);
        else
            Error("Undefined Intersection");
    }
    else if (Poly* p1 = dynamic_cast<Poly*>(&lhs))
    {
        if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
            DoHatchArea2(*p2, *p1);
        else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs))
            DoHatchArea4(*p2, *p1);
        else if (Poly* p2 = dynamic_cast<Poly*>(&rhs))
            DoHatchArea6(*p1, *p2);
        else
            Error("Undefined Intersection");
    }
    else
    {
        Error("Undefined Intersection");
    }
}
```

哇喔，还真不少！可以看出，这种蛮干的做法要求你编写大量（而且琐碎的）代码。好处是任何一位称职的 C++ 程序员都可以写出这么一大堆 if 语句。此外还有一个优点：如果 classes 数目不太多，代码执行速度会很快。从速度的角度来看，DoubleDispatch() 在这一整组可能的型

别中采用线性查找。由于这个动作并非“往覆式”（只是一组连续而非循环的 if-else 语句），对小型集合而言速度很快。

这个暴力法存在一个问题，纯粹属于代码数量上的问题：随着 classes 数目的增长，代码将愈来愈难维护。上面只不过处理了三个 classes，已经需要相当可观的程序量。如果增加更多 classes，代码的数量将呈指数增长。设想一下，如果面临“内含 20 个 classes”的继承体系，DoubleDispatch() 会是什么样子！

另一个问题在于，DoubleDispatch() 是依存性 (dependency) 上的瓶颈——它的实作必须首先知晓继承体系中的所有 classes。依存关系当然愈松散愈好，而 DoubleDispatch() 却是吞噬依存性的巨兽。

DoubleDispatch() 存在的第三个问题是，if 语句的次序很具关键。这是一个十分隐微而危险的问题。举个例子，假设你从 Rectangle 派生了一个 RoundedRectangle（弧角矩形），而后修改 DoubleDispatch()：在所有 if-else 语句之后并恰在 Error() 调用之前，插入新增的 if 语句。这样的话，你就引入了一只“臭虫”。

原因是，如果向 DoubleDispatch() 传递一个 pointer to RoundedRectangle，dynamic_cast<Rectangle*> 会成功。该测试位于 dynamic_cast<RoundedRectangle*> 测试之前。因此，第一个测试会“吃掉”Rectangles 和 RoundedRectangle。这使得第二个测试永远没有被执行的机会。大多数编译器对此不会发出警告。

或许可以像下面这样修改测试句：

```
void DoubleDispatch(Shape& lhs, Shape& rhs)
{
    if (typeid(lhs) == typeid(Rectangle))
    {
        Rectangle* p1 = dynamic_cast<Rectangle*>(&lhs);
        ...
    }
    else ...
}
```

现在，这些测试针对的是“确切的型别”，而不是“确切型别或派生型别”。上述代码中，如果 lhs 是 RoundedRectangle，typeid 比较式将会失败，测试会继续向下执行。最终，对 typeid(RoundedRectangle) 的测试会成功。

哎呀，这根本是顾此失彼：如今的 DoubleDispatch() 变得太死板了。要知道，这个函数如果没有支持某一型别，你便会希望它作用于最相近的基础型别 (base type) 上。这正是你使用继承时通常想要的效果——缺省情况下派生对象应该像基础对象那样地运作，除非你改写了某个行为。问题是，在这个基于 typeid 之上的 DoubleDispatch() 版本中，这样的特性无法维持。这里得出的经验是：在 DoubleDispatch() 中你还是得使用 dynamic_cast，并且还得对 if 测试语句“排序”，使最底层派生类 (most derived classes) 首先被测试。

这就给“multimethods 暴力法”带来了另外两个缺点。首先，`DoubleDispatch()` 和 `Shape` 继承体系之间的依存性加深了——`DoubleDispatch()` 不仅必须知道 `classes`，还必须知道 `classes` 之间的继承关系。其次，由于必须维护动态转型（dynamic casts）的正确次序，维护者肩上的担子更重了。

11.4 将暴力法自动化

在某些场合，暴力法的速度无与伦比，因此，值得我们认真实现这样一种分派手法（dispatcher）。此时运用 `typelists` 将大有帮助。

回顾第 3 章，Loki 定义了一种 `typelist` 工具——一个由结构（structure）和编译期算法（compile-time algorithms）组成的集合，用来帮助你操纵“型别集”。在 `multimethods` 暴力法中，我们可以使用客户提供的 `typelist`，其中每个 `types` 代表的是继承体系中的 `classes`（本例即为 `Rectangle`、`Poly`、`Ellipse`，等等），然后以一个递归模板（recursive template）产生一系列的 `if-else` 语句。

一般情况下，“分派”可以建立在不同的“型别集”上，所以左操作数（operand）的 `typelist` 可以不同于右操作数的 `typelist`。

下面我大致写出一个 `StaticDispatcher` class template；其中会执行型别推导（type deduction）算法，最后调用另一个 `class` 中的一个函数。稍后另有代码说明。

```
template
<
    class Executor,
    class BaseLhs,
    class BaseRhs = BaseLhs,
    class TypesRhs = TypesLhs,
    typename ResultType = void
>
class StaticDispatcher
{
    typedef typename TypesLhs::Head Head;
    typedef typename TypesLhs::Tail Tail;
public:
    static ResultType Go(BaseLhs& lhs, BaseRhs& rhs,
        Executor exec)
    {
        if (Head* p1 = dynamic_cast<Head*>(&lhs)) {
            return StaticDispatcher<Executor, BaseLhs,
                NullType, BaseRhs, TypesRhs>::DispatchRhs(
                *p1, rhs, exec);
        }
        else {
            return StaticDispatcher<Executor, BaseLhs,
                Tail, BaseRhs, TypesRhs>::Go(lhs, rhs, exec);
        }
    }
}
```

```
...
};
```

如果你熟悉 `typelists`, `StaticDispatcher()` 的工作方式对你而言应该很简单。相对于其所完成的工作, `StaticDispatcher()` 代码少得惊人。它共有六个 `template` 参数, 其中 `Executor` 负责实际工作 (亦即本例的“绘制相交区影线”)。稍后我将深入介绍 `Executor`。

`BaseLhs` 和 `BaseRhs` 分别是左侧引数和右侧引数的基础型别。`TypesLhs` 和 `TypesRhs` 都是 `typelists`, 包含上述两引数的可能派生型别。如果 `BaseRhs` 和 `TypesRhs` 采用缺省值, 我们得到的 `dispatcher` (分派器) 针对的将是“位于同一个 `class` 继承体系内”的型别, 这正是先前绘图程序的情况。

`ResultType` 是 `double-dispatch` 操作的返回型别。一般情况下被分派 (最终被调用) 的函数可以传回任意型别。`StaticDispatcher()` 支持这种一般性, 并将结果转发给调用者。

`StaticDispatcher::Go` 试图从“lhs 的地址”着手, 将其动态转型为“`TypesLhs` `typelist` 中的第一个型别”。如果动态转型失败, 会递归调用 `Go`, 并赋予 `TypeLhs` 剩余部分 (即 `tail` 部分)。这不是真正的递归调用, 因为我们每次获得的并不是相同的 `StaticDispatcher` 具现体。

净效应是, `Go` 将执行一组 `if-else` 语句, 对 `typelist` 中的每一个型别施行 `dynamic_cast`。如果找到匹配的型别, `Go` 将调用 `DispatchRhs()` ——由它执行第二步 (也是最后一步) 型别推导: 寻找 `rhs` 的动态型别。

```
template <...>
class StaticDispatcher
{
    ... as above ...
    template <class SomeLhs>
    static ResultType DispatchRhs(SomeLhs& lhs, BaseRhs& rhs,
                                   Executor exec)
    {
        typedef typename TypesRhs::Head Head;
        typedef typename TypesRhs::Tail Tail;

        if (Head* p2 = dynamic_cast<Head*>(&rhs))
        {
            return exec.Fire(lhs, *p2);
        }
        else
        {
            return StaticDispatcher<Executor, SomeLhs,
                                   NullType, BaseRhs, Tail>::DispatchRhs(
                                   lhs, rhs, exec);
        }
    }
};
```

`DispatchRhs()` 对 `rhs` 施行相同的算法，就像 `Go` 对付 `lhs` 一样。如果对 `rhs` 动态转型成功，`DispatchRhs()` 会调用 `Executor::Fire`，并将找到的两个型别传给它。同样，编译器生成的一组由 `if-else` 语句组成的代码。有趣的是针对 `TypesLhs` 内的每一个型别，编译器都会生成这样一组 `if-else` 语句。经由两个 `typelists` 和一个固定的程序库，`StaticDispatcher` 实际上产生了指数级数量的代码。这当然很有价值，但也有潜在危险——过多代码会影响编译时间、程序大小、执行时间。

为了处理“递归停止条件”，我们需要针对两种情况来特化（specialize）`StaticDispatcher`：(1) 未能在 `TypesLhs` 中找到 `lhs`，(2) 未能在 `TypesRhs` 中找到 `rhs`。

如果将 `NullType` 当做 `TypesLhs`，然后对 `StaticDispatcher` 调用 `Go`，第一种情况（在 `lhs` 上出错）就会出现。这是 `TypesLhs` 查找失败的表现。（回顾第 3 章，`NullType` 用来表示“任何 `typelist` 中的最后一个元素”）

```
template
<
    class Executor,
    class BaseLhs,
    class BaseRhs,
    class TypesRhs,
    typename ResultType
>
class StaticDispatcher<Executor, BaseLhs, NullType,
    BaseRhs, TypesRhs, ResultType>
{
    static void Go(BaseLhs& lhs, BaseRhs& rhs, Executor exec)
    {
        exec.OnError(lhs, rhs);
    }
};
```

错误处理（error handling）被漂亮地委派（delegated）给 `Executor` class；稍后在关于 `Executor` 的讨论中，你会进一步看到这一点。

如果将 `NullType` 当做 `TypesRhs`，然后对 `StaticDispatcher` 调用 `DispatchRhs`，第二种情况（在 `rhs` 上出错）就会出现。因此，需要下面这样一个特化版：

```
<
    class Executor,
    class BaseLhs,
    class TypesLhs,
    class BaseRhs,
    class TypesRhs,
    typename ResultType
>
class StaticDispatcher<Executor, BaseLhs, TypesLhs,
    BaseRhs, NullType, ResultType>
{
public:
```



```
static void DispatchRhs(BaseLhs& lhs, BaseRhs& rhs,
    Executor& exec)
{
    exec.OnError(lhs, rhs);
}
};
```

是时候了，现在要讨论，为利用上述定义的 double-dispatch 引擎，Executor 应当如何实现。

StaticDispatcher 只负责查找型别。正确的型别和对象找到之后，会被传给 Executor::Fire()。为区分这么多调用动作，Executor 必须重载多个 Fire()。例如在“图形相交区绘制影线”的例子中，其 Executor class 大致如下：

```
class HatchingExecutor
{
public:
    // Various intersection algorithms
    void Fire(Rectangle&, Rectangle&);
    void Fire(Rectangle&, Ellipse&);
    void Fire(Rectangle&, Poly&);
    void Fire(Ellipse&, Poly&);
    void Fire(Ellipse&, Ellipse&);
    void Fire(Poly&, Poly&);

    // Error handling routine
    void OnError(Shape&, Shape&);
};
```

你可以像下面代码那样，将 HatchingExecutor 用于 StaticDispatcher：

```
typedef StaticDispatcher<HatchingExecutor, Shape,
    TYPELIST_3(Rectangle, Ellipse, Poly)> Dispatcher;
Shape *p1 = ...;
Shape *p2 = ...;
HatchingExecutor exec;
Dispatcher::Go(*p1, *p2, exec);
```

这段代码调用了“HatchingExecutor class 中合适的 Fire() 重载版本”。你可以将 StaticDispatcher class template 视为一种获得“动态重载”（亦即将重载规则推迟至执行期）的机制。这使得 StaticDispatcher 非常容易被使用。实作 HatchingExecutor 时，你只需将重载规则记在心里，然后将 StaticDispatcher 当做黑盒子来用；StaticDispatcher 自会在执行期施展魔法，实施重载规则。

这里有一个不错的副作用：StaticDispatcher 会在编译期发现任何重载歧义（overloading ambiguities）。举个例子，假设你没有声明 HatchingExecutor::Fire(Ellipse&, Poly&)，但声明了 HatchingExecutor::Fire(Ellipse&, Shape&) 和 HatchingExecutor::Fire(Shape&, Poly&)。那么通过一个 Ellipse 和一个 Poly 来调用 HatchingExecutor::Fire() 会导致歧义（模棱两可）——两个函数都会抢着处理这个调用。值得注意的是，StaticDispatcher 会为你指出这个错误，并带有同样详细的错误消息。StaticDispatcher

的确是一种“和现有 C++ 重载规则十分一致”的工具。

如果出现执行期错误——例如你将 `Circle` 作为一个引数传给 `StaticDispatcher::Go`——会发生什么事？一如先前所提示，处理边界情况时 `StaticDispatcher` 只不过调用 `Executor::OnError` 而已，而且调用时使用的是原始的（而非转型后的）`lhs` 和 `rhs`。也就是说，在本例之中，`HatchingExecutor::OnError(Shape&, Shape&)` 就是错误处理函数。利用它你可以做任何你认为合适的事情。一旦它被调用，便意味着 `StaticDispatcher` 放弃了寻找动态型别的努力。

如同前一节所说，继承（inheritance）会给暴力式分派器（dispatcher）带来问题。也就是说下面的 `StaticDispatcher` 具体体内有一只“臭虫”：

```
typedef StaticDispatcher
<
    SomeExecutor,
    Shape,
    TYPELIST_4(Rectangle, Ellipse, Poly, RoundedRectangle)
>
MyDispatcher;
```

如果将 `RoundedRectangle` 传给 `MyDispatcher`，它会被当做 `Rectangle`。这导致“对 pointer to `RoundedRectangle` 施行 `dynamic_cast<Rectangle*>`”将会成功，而且由于 `dynamic_cast<RoundedRectangle*>` 位于“食物链”下层，所以后者永远没有被调用的机会。正确写法应当是：

```
typedef StaticDispatcher
<
    SomeExecutor,
    Shape,
    TYPELIST_4(RoundedRectangle, Ellipse, Poly, Rectangle)
>
Dispatcher;
```

一般原则是，将最底层派生型别（most derived types）放在 `typelist` 的头端。

如果能够在代码中自动运用这条原则，那就太好了；`typelist` 的确支持这一点。我们有办法在编译期检测继承关系（第 2 章），而 `typelist` 可被排序。正是这样导致了第 3 章的 `DerivedToFront` 编译期算法。

为使用自动排序功能，我们只需这样修改 `StaticDispatcher`：

```
template <...>
class StaticDispatcher
{
    typedef typename DerivedToFront<
        typename TypesLhs::Head>::Result Head;
    typedef typename DerivedToFront<
        typename TypesLhs::Tail>::Result Tail;
public:
    ... as above ...
};
```

不要忘了，在这一切自动化之后，我们得到的只不过是代码生成方面的好处，依存问题依然存在。`StaticDispatcher` 的确使得暴力式 `multimethods` 很容易实现，但它还是依存于继承体系中的所有型别。它的优势在于速度（如果继承体系中没有太多型别）和非侵入性（*nonintrusiveness*）——在某继承体系中使用 `StaticDispatcher` 时，你不必修改该继承体系。

11.5 暴力式 Dispatcher 的对称性

为两种形状的相交区绘制影线时，对于“矩形覆盖于椭圆上”和“椭圆覆盖于矩形上”这两种不同情况，你可能希望绘制方法有所不同，但也可能你希望无论谁覆盖在谁之上，都以相同的方式为相交区绘制影线。后一种情况你需要所谓的 *symmetric multimethod*——也就是引数的传递顺序无关紧要。

对称性只适用于“两个参数的型别完全相同”的情况（在我们的例子中也就是 `BaseLhs` 和 `BaseRhs` 相同，`LhsTypes` 也和 `RhsTypes` 相同。）

前面定义的暴力式 `StaticDispatcher` 是非对称性的：也就是说，它并没有为对称式的 `multimethods` 提供任何内建支持。举个例子，假设你定义了下面的 `classes`：

```
class HatchingExecutor
{
public:
    void Fire(Rectangle&, Rectangle&);
    void Fire(Rectangle&, Ellipse&);
    ...
    // Error handler
    void OnError(Shape&, Shape&);
};

typedef StaticDispatcher
<
    HatchingExecutor,
    Shape,
    TYPELIST_3(Rectangle, Ellipse, Poly)
>
HatchingDispatcher;
```

如果将 `Ellipse` 当做左侧参数传递，将 `Rectangle` 当做右侧参数传递，`HatchingDispatcher` 是不会动作的。尽管从 `HatchingExecutor` 的角度来看，谁前谁后没有关系，但它还是坚持：你必须以一定的次序传递对象。

我们可以在客户代码中实现对称性，做法是：颠倒参数次序，并将调用动作从一个重载版本转派（*forwarding*）至另一个重载版本：

```
class HatchingExecutor
{
public:
```

```

void Fire(Rectangle&, Ellipse&);
// Symmetry assurance
void Fire(Ellipse& lhs, Rectangle& rhs)
{
    // Forward to Fire(Rectangle&, Ellipse&)
    //   by switching the order of arguments
    Fire(rhs, lhs);
}
...
};

```

这些小型转派函数（forwarding functions）不好维护。理想情况下 `StaticDispatcher` 应该增加一个 `bool template` 参数，藉以提供可选择的对称性支持；这一点值得深入讨论。

我们需要做的是：调用 `callback`（回调函数）时，针对某些情况，`StaticDispatcher` 需要颠倒引数次序。哪些情况呢？我们来分析前一个例子。根据 `template` 引数列表的缺省值，扩展开来之后，我们得到下面的具体体（instantiation）：

```

typedef StaticDispatcher
<
    HatchingExecutor,
    Shape,
    TYPELIST_2(Rectangle, Ellipse, Poly), // TypesLhs
    Shape,
    TYPELIST_2(Rectangle, Ellipse, Poly), // TypesRhs
    void
>
HatchingDispatcher;

```

为对称式 dispatcher 选择参数对组（parameter pairs）时，一个可用的算法是：将第一个 `typelist`（即 `TypesLhs`）中的第一个型别和第二个 `typelist`（即 `TypesRhs`）中的每一个型别组合起来。这为本例带来三对组合：`Rectangle-Rectangle`、`Rectangle-Ellipse`、`Rectangle-Poly`。接着，将 `TypesLhs` 中的第二个型别（即 `Ellipse`）和 `TypesRhs` 中的所有型别组合起来，但因为第一对组合（`Rectangle-Ellipse`）已经在第一个步骤中得到了，所以这一次从 `TypesRhs` 的第二个元素开始。这次产生了 `Ellipse-Ellipse` 和 `Ellipse-Poly`。同样的做法适用于下一步骤：`TypesLhs` 的 `Poly` 只能和 `TypesRhs` 内的第三个元素之后的所有元素组合。这次只产生一个组合：`Poly-Poly`；至此，算法结束。

采用这一算法，你只需针对选择好的组合来实现各个函数，像下面这样：

```

class HatchingExecutor
{
public:
    void Fire(Rectangle&, Rectangle&);
    void Fire(Rectangle&, Ellipse&);
    void Fire(Rectangle&, Poly&);
    void Fire(Ellipse&, Ellipse&);

```

```

void Fire(Ellipse&, Poly&);
void Fire(Poly&, Poly&);
// Error handler
void OnError(Shape&, Shape&);
};

```

至于上述算法所排除的组合, 包括 `Ellipse-Rectangle`、`Poly-Rectangle` 和 `Poly-Ellipse`, `StaticDispatcher` 必须完全靠自己检测。针对这三种组合, `StaticDispatcher` 必须颠倒引数次序。对于其他所有组合, 则像以前一样地将调用转派 (forwarding) 出去。

“是否需要调引数”的决定条件是什么? 在 `TL2` 中, 这个算法所选择的型别, 其索引都“大于等于”`TL1` 中的型别的索引。因此, 判断条件如下:

对于两个型别 `T` 和 `U`, 如果 `U` 在 `TypesRhs` 中的索引小于 `T` 在 `TypesLhs` 中的索引, 那么, 引数次序必须对调。

假设 `T` 为 `Ellipse`, `U` 为 `Rectangle`。那么 `T` 在 `TypesLhs` 中的索引是 1, `U` 在 `TypesRhs` 的索引是 0。所以在调用 `Executor::Fire` 之前, `Ellipse` 和 `Rectangle` 必须对调: 这是正确的。

`typelist` 工具已经提供了一个编译期算法 `IndexOf`, 这个算法传回的是“某一型别在 `typelist` 中的位置”, 因此, 我们可以很方便地表达对调条件。

首先我们必须增加一个新的 `template` 参数, 用来表明这个 `dispatcher` 是否对称, 然后再增加一个小巧的 `traits class template: InvocationTraits`。调用 `Executor::Fire` 时, `InvocationTraits` 要么对调引数, 要么就不对调。以下是部分代码。

```

template
<
    class Executor,
    bool symmetric,
    class BaseLhs,
    class TypesLhs,
    class BaseRhs = BaseLhs,
    class TypesRhs = TypesLhs,
    typename ResultType = void
>
class StaticDispatcher
{
    template <bool swapArgs, class SomeLhs, class SomeRhs>
    struct InvocationTraits
    {
        static void DoDispatch(SomeLhs& lhs, SomeRhs& rhs,
                               Executor& exec)
        {
            exec.Fire(lhs, rhs);
        }
    };
    template <class SomeLhs, class SomeRhs>
    struct InvocationTraits<True, SomeLhs, SomeRhs>
    {

```

```

        static void DoDispatch(SomeLhs& lhs, SomeRhs& rhs,
                               Executor& exec)
        {
            exec.Fire(rhs, lhs); // swap arguments
        }
    }
public:
    static void DispatchRhs(BaseLhs& lhs, BaseRhs& rhs,
                           - Executor exec)
    {
        if (Head* p2 = dynamic_cast<Head*>(&rhs))
        {
            enum { swapArgs = symmetric &&
                   IndexOf<Head, TypesRhs>::result <
                   IndexOf<BaseLhs, TypesLhs>::result };
            typedef InvocationTraits<swapArgs, BaseLhs, Head>
                CallTraits;
            return CallTraits::DoDispatch(lhs, *p2);
        }
        else
        {
            return StaticDispatcher<Executor, BaseLhs,
                                   NullType, BaseRhs, Tail>::DispatchRhs(
                lhs, rhs, exec);
        }
    }
};

```

对称性支持会对 `StaticDispatcher` 带来某些复杂性，但对 `StaticDispatcher` 用户来说，生活当然变得更为方便。

11.6 对数型 (Logarithmic) Double Dispatcher

如果想避免暴力式解法导致的严重依存性，你得寻求更动态的解法。你不能在编译期生成代码，必须维护某个执行期结构，并使用执行期算法，从而有助于“根据型别，动态分派函数调用”。

在这里，RTTI（执行期型别辨识）可以提供进一步帮助，因为它不仅提供了 `dynamic_cast` 和型别标识，还通过 `std::type_info` 的 `before()` 函数提供了执行期型别次序。`before()` 为程序中的所有型别提供了次序关系。利用这种次序关系，我们可以对型别进行快速查找。

这里的做法类似于 Scott Meyers 在《*More Effective C++*》（1996a）条款 31 的做法，但有一些改进：调用处理函数（handler）时，转型步骤得以自动化，而且这里的目标还包括泛型化。

我们将利用第 2 章介绍的 `OrderedTypeInfo` class。这是一个 wrapper（外覆类），它提供完全相同于 `std::type_info` 的功能。此外还提供“value 语义”和一个无预警的 `less` 操作符。因此你可以将 `OrderedTypeInfo` 对象存放于标准容器中——这一点对本章来说很重要。

Meyers 的做法很简单：针对“你希望据以分派函数”的每一对 (pair) `std::type_info` 对象，你都可以在 `double dispatcher` 中注册一个函数指针。这个 `double dispatcher` 将信息存储于一个 `std::map` 中。执行期间，当通过两个未知对象调用 `double dispatcher` 时，这个 `double dispatcher` 会执行一次快速查找（对数级时间）找出型别。如果找到目标，就触发相应的函数指针。

现在我们来定义一个泛型引擎 (`generic engine`) 结构，这个引擎运用以上原则。我们必须经由两个引数（左侧引数和右侧引数）的基础型别来模板化 (`templatize`) 这个引擎。我们将这个引擎称为 `BasicDispatcher`，因为它将成为稍后数个更高级的 `double dispatcher` 的基础设备。

```
template <class BaseLhs, class BaseRhs = BaseLhs,
         typename ResultType = void>
class BasicDispatcher
{
    typedef std::pair<OrderedTypeInfo, OrderedTypeInfo>
        KeyType;
    typedef ResultType (*CallbackType)(BaseLhs&, BaseRhs&);
    typedef CallbackType MappedType;
    typedef std::map<KeyType, MappedType> MapType;
    MapType callbackMap_;
public:
    ...
};
```

在这个 `map` 中，`key` 的型别是 `std::pair`，由两个 `OrderedTypeInfo` 对象组成。`std::pair` class 是带序的 (`ordered`)，所以我们不必提供用户自定义的排序仿函数 (`ordering functor`)。

如果将 `callback` 的型别模板化，`BasicDispatcher` 将更具一般性。一般来说，`callback` 不一定是函数，它也可以是一个仿函数（参阅第 5 章引介部分，那里有仿函数的介绍）。如果将内部的“`CallbackType` 型别定义”转换为一个模板参数，`BasicDispatcher` 就可以接受仿函数。

另一个重要改进是：不使用 `std::map`，改用更有效率的结构。Matt Austern (2000) 曾说过，标准关联式容器 (`standard associative containers`) 的应用领域比你想象的要窄一些。特别是在空间和时间方面，一个结合二分查找法（例如 `std::lower_bound`）的 `sorted vector` 会比关联式容器表现更好。当访问次数远多于安插次数的时候，这种情况就会发生，所以我们应当细致地分析 `double-dispatcher` 对象的典型使用模式。

通常 `double dispatcher` 是一种“写一次，读多次”的结构。一般来说，一个程序只需要设置 `callbacks` 一次，然后使多次使用这个 `dispatcher`。这与虚函数机制一致，而这正是 `double dispatchers` 所要扩充的。在编译期，你可以决定哪些函数是虚函数，哪些不是。

看来我们得使用 `sorted vector`。此物的缺点在于安插和删除都是线性时间，然而 `double dispatcher` 通常并不在意这两个操作的速度。从另一方面说，`vector` 提供大约两倍的查询速度并耗用小得多的工作区，因而对 `double dispatcher` 来说，`sorted vector` 无疑是更好的选择。

Loki 定义了一个 `AssocVector class template`，可为你免除手工维护 `sorted vector` 的烦恼。

AssocVector 是 `std::map` 的替代物（它们支持同一组成员函数），并基于 `std::vector` 之上。**AssocVector** 不同于 `map` 之处在于 `erase()` 函数的行为（`AssocVector::erase` 会使得所有指向该对象的迭代器失效），以及 `insert()` 和 `erase()` 的复杂度保证（线性，而非常数）。由于 **AssocVector** 和 `std::map` 高度兼容，所以对于 `double dispatcher` 内含的那份数据结构，我将继续以 `map` 称呼之。

以下是修改后的 **BasicDispatcher** 定义：

```
template
<
    class BaseLhs,
    class BaseRhs = BaseLhs,
    typename ResultType = void,
    typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
>
class BasicDispatcher
{
    typedef std::pair<TypeInfo, TypeInfo>
        KeyType;
    typedef CallbackType MappedType;
    typedef AssocVector<KeyType, MappedType> MapType;
    MapType callbackMap_;
public:
    ...
};
```

注册函数（**registration function**）很容易定义。下面就是我们所需要的：

```
template <...>
class BasicDispatcher
{
    ... as above ...
    template <class SomeLhs, SomeRhs>
    void Add(CallbackType fun)
    {
        const KeyType key(typeid(SomeLhs), typeid(SomeRhs));
        callbackMap_[key] = fun;
    }
};
```

SomeLhs 和 **SomeRhs** 是两个具体型别；正是为了这两个型别，你需要分派（**dispatch**）调用动作。和 `std::map` 一样，**AssocVector** 重载了 `operator[]`，用来寻找 *key* 所对应的型别。如果没有找到目标，就会加入一个新元素。`operator[]` 会传回一个 **reference**，指向“新增的”或“被找到的”元素，而后 `Add()` 会将 `fun` 指派给该元素。

以下是 `Add()` 的使用范例：

```
typedef BasicDispatcher<Shape> Dispatcher;
// Hatches the intersection between a rectangle and a polygon
void HatchRectanglePoly(Shape& lhs, Shape& rhs)
{
    Rectangle& rc = dynamic_cast<Rectangle&>(lhs);
    Poly& pl = dynamic_cast<Poly&>(rhs);
```



```

    ... use rc and pl ...
}
...
Dispatcher disp;
disp.Add<Rectangle, Poly>(HatchRectanglePoly);

```

执行查找 (search) 和呼唤 (invocation) 动作的成员函数很简单:

```

template <...>
class BasicDispatcher
{
    ... as above ...
    ResultType Go(BaseLhs& lhs, BaseRhs& rhs)
    {
        MapType::iterator i = callbackMap_.find(
            KeyType(typeid(lhs), typeid(rhs)));
        if (i == callbackMap_.end())
        {
            throw std::runtime_error("Function not found");
        }
        return (i->second)(lhs, rhs);
    }
};

```

11.6.1 对数型 (Logarithmic) Dispatcher 和继承 (Inheritance)

面对继承, `BasicDispatcher` 无法正常运作。在 `BasicDispatcher` 中如果你只注册了 `HatchRectanglePoly(Shape& lhs, Shape& rhs)`, 那么, 你就只能为 `Rectangle` 对象和 `Poly` 对象提供正确的分派操作, 其他种对象都不行。例如你将 `RoundedRectangle` 和 `Poly` 的 references 传给 `BasicDispatcher::Go()`, 将会被拒绝。

这样的行为不符合继承规则; 根据继承规则, 缺省情况下, 派生型别的行为必须和基础型别的行为一样。如果 `BasicDispatcher` 也接受“以 `derived classes` 对象为参数”的调用——只要这些调用以 C++ 重载规则看来没有歧义 (并非模棱两可)——那就太好了。

你可以做点事情来纠正这一问题, 但截至目前, 完整方案尚不存在。在 `BasicDispatcher` 中你必须细心地注册所有型别对组 (pairs of types)³⁹。

11.6.2 对数型 (Logarithmic) Dispatcher 和转型 (Casts)

`BasicDispatcher` 可用, 但不完美。你可以注册一个函数来处理 `Rectangle` 和 `Poly` 的相交问题, 但那个函数必须以基础型别 `Shape&` 为引数。可要求客户端 (`HatchRectanglePoly` 实作端) 将 `Shape&` 转回正确型别, 不但笨拙而且容易产生错误。

另一方面, `callback map` 无法为每一个元素保存不同的函数或仿函数型别, 所以, 我们必须坚持一致的表示法。《*More Effective C++*》(Meyers 1996a) 条款 31 也讨论了这个问题。“函数指针

³⁹ 我相信一定有解决“继承问题”的方案。但是很遗憾, 作家也有最后期限的问题。

至函数指针”的转型在此无济于事，因为退出 `FnDoubleDispatcher::Add` 之后，静态型别信息已经丢失，你不知道该转为什么型别。（如果这听起来让你觉得迷惑，请试着写一些代码，你就立刻知道是怎么回事了）

以下我将实现一个方案，用以解决这一转型问题。此方案以简单的 `callback` 函数（而非仿函数）为基础，也就是说，`template` 引数 `CallbackType` 是一个函数指针。

一个可能带来帮助的想法是：使用所谓的 **trampoline function**（“弹簧垫”函数），又称为 *thunk*（形实转换函数）。这是一种小型函数，用来在调用其他函数之前作一些小调整。C++ 编译器设计者经常使用它们来实现某些功能，例如实现 `covariant return types`（译注：可参考《The C++ Programming Language》15.6.2 节）、为多重继承（multiple inheritance）调整指针，等等。

我们可以藉由 **trampoline function** 来执行合适的转换，然后调用具有正确原型的函数，从而使客户获得方便。问题是现在 `callbackMap_` 必须保存两个函数指针：一个是客户提供的函数指针，另一个是 **trampoline function** 指针。在速度方面，这让人忧心。我们面对的不是“一个”经由指针的间接调用，而是“两个”。此外，实作上会变得更复杂。

一个有趣的小手法可以让你柳暗花明。函数指针可被当做“`nontype template` 参数”来接受（在本书中“`nontype template` 参数”通常是整数值）。“指向全局对象（包括函数）”的指针亦可被当做“`nontype template` 参数”来接受。唯一条件是，如果某函数的地址被当做 `template` 参数，该函数必须具备外部连接（`external linkage`）。在具备外部连接的函数中，你可以很容易地转换静态函数，只要去除关键字 `static`，并将函数放在匿名命名空间（`unnamed namespaces`）。例如，在未支持命名空间的 C++ 中这么写：

```
static void Fun();
```

可在匿名命名空间中改成这样：

```
namespace
{
    void Fun();
}
```

将函数指针当做“`nontype template` 参数”，意味着我们不再需要利用 `map` 来保存它。这是很重要的一点，需要透彻理解。之所以不再需要保存这个函数指针，原因是编译器知道它的静态信息。因此编译器可以将函数地址写死在 **trampoline function** 之内。

我们在新的 `class` 中实现这一想法，这个 `class` 将 `BasicDispatcher` 作为其后端（`back end`）。新 `class` 称为 `FnDispatcher`，经过调整的它只会将调用动作分派至“函数”而非“仿函数”。`FnDispatcher` 以 `private` 方式聚合（`aggregates`）了 `BasicDispatcher`，并提供相应的转派函数（`forwarding function`）。

`FnDispatcher::Add()` 接受三个 `template` 参数。前两个表示分派操作中的左侧和右侧型别（`ConcreteLhs` 和 `ConcreteRhs`）。第三个参数（`callback`）是个函数指针。新增的

`FnDispatcher::Add` 重载了先前定义过的那个带有两个 `template` 参数的 `Add` 函数。

```
template <class BaseLhs, class BaseRhs = BaseLhs,
         ResultType = void>
class FnDispatcher
{
    BaseDispatcher<BaseLhs, BaseRhs, ResultType> backEnd_;
    ...
public:
    template <class ConcreteLhs, class ConcreteRhs,
              ResultType (*callback)(ConcreteLhs&, ConcreteRhs&)>
    void Add()
    {
        struct Local // see Chapter 2
        {
            static ResultType Trampoline(BaseLhs& lhs, BaseRhs& rhs)
            {
                return callback(
                    dynamic_cast<ConcreteLhs&>(lhs),
                    dynamic_cast<ConcreteRhs&>(rhs));
            }
        };
        return backEnd_.Add<ConcreteLhs, ConcreteRhs>(
            &Local::Trampoline);
    }
};
```

运用一个局部结构，我们就在 `Add()` 之中定义了 `trampoline`，将引数转换为正确型别，然后转至 `callback`。接着 `Add()` 函数再使用 `backEnd_` 的 `Add()` 函数（由 `BaseDispatcher` 定义），将 `trampoline` 添加到 `callbackMap_` 内。

`trampoline function` 并不会增加速度上的开销。对 `callback` 的调用，看起来像是一个间接调用，其实不然。前面说过，编译器会将 `callback` 的地址写死于 `Trampoline` 中，所以不存在第二层间接性。可能的话，聪明的编译器甚至会将调用 `callback` 的动作 `inline` 化。

新定义的 `Add` 函数用起来很简单：

```
typedef FnDispatcher<Shape> Dispatcher;

// Possibly in an unnamed namespace
void HatchRectanglePoly(Rectangle& lhs, Poly& rhs)
{
    ...
}

Dispatcher disp;
disp.Add<Rectangle, Poly, Hatch>();
```

由于 `Add()` 函数, `FnDispatcher` 很容易使用。`FnDispatcher` 也有一个 `Add()` 函数, 和 `BaseDispatcher` 定义的那个类似: 所以如果需要的话, 你也可以使用这个函数⁴⁰。

11.7 `FnDispatcher` 和对称性

`FnDispatcher` 具有动态性, 所以, 为它添加对称性就比为“静态的 `StaticDispatcher`”添加对称性要简单得多。

为支持对称性, 我们只需注册两个 trampolines: 一个以正常顺序调用执行者 (executor), 另一个在调用之前先对调参数次序。我们为 `Add` 增加一个新的 `template` 参数如下。

```
template <class BaseLhs, class BaseRhs = BaseLhs,
         typename ResultType = void>
class FnDispatcher
{
    ...
    template <class ConcreteLhs, class ConcreteRhs,
              ResultType (*callback)(ConcreteLhs&, ConcreteRhs&),
              bool symmetric>
    bool Add()
    {
        struct Local
        {
            ... Trampoline as before ...
            static void TrampolineR(BaseRhs& rhs, BaseLhs& lhs)
            {
                return Trampoline(lhs, rhs);
            }
        };
        Add<ConcreteLhs, ConcreteRhs>(&Local::Trampoline);
        if (symmetric)
        {
            Add<ConcreteRhs, ConcreteLhs>(&Local::TrampolineR);
        }
    }
};
```

`FnDispatcher` 的对称性处于“函数”层次——对于你注册的每一个函数, 你都可以决定是否需要对称性分派 (symmetric dispatching)。

11.8 Double Dispatch (双重分派) 至仿函数 (Functors)

前面说过, trampoline 技术对非静态函数指针非常有效。藉助匿名命名空间, 我们可以运用一种干净的手法, 以“当前编译单元之外不可见”的非静态函数, 代替静态函数。

⁴⁰ 这种情况下你不能使用 `FnDispatcher::Add()`: 当你需要动态注册被装载的函数。但即使在这种情况下, 只要对设计作些微小修改, 你还是可以利用 trampoline。

但有时候,你需要的是更实在的 `callback` 对象(`BasicDispatcher` 的 `CallbackType` 模板参数),而非简单的函数指针。例如你可能希望每个 `callback` 都保存某些状态 (`state`),而函数无法保存太多状态 (只能保存一些静态变量)。此时你需要在 `double dispatcher` 中注册仿函数,而不是函数。

仿函数 (第 5 章) 是一种 `class`, 重载了 “function call 操作符” `operator()`, 从而在调用语法上模拟了一般函数。此外,仿函数还可以通过成员变量来保存和访问状态。遗憾的是, `trampoline` 技术之所以对仿函数不起作用,原因正在于:仿函数保存了状态,而一般函数没有 (trampoline 会在哪里保存状态?)

客户端可通过合适的仿函数型别,具现出 `BasicDispatcher`, 从而直接使用它。

```
struct HatchFuncor
{
    void operator()(Shape&, Shape&)
    { ... }
};

typedef BasicDispatcher<Shape, Shape, void, HatchFuncor>
    HatchingDispatcher;
```

`HatchFuncor::operator()` 本身无法成为虚函数,因为 `BasicDispatcher` 需要的是具有 “value 语义” 的仿函数,而 “value 语义” 和执行期多态是无法和睦共处的。但是,唔, `HatchFuncor::operator()` 可以将调用转派至虚函数。

真正的缺点是, `dispatcher` 本可提供的某些自动化行为,如转型处理、提供对称性等等,客户都无法得到。

好像我们又回到了原点。但如果阅读过第 5 章关于泛化仿函数的讨论,你就不会这么想。第 5 章定义了一个名为 `Functor` 的 `class template`, 它能聚集 (aggregate) 任何种类的仿函数、函数指针,甚至另一个 `Functor` 对象。经由 `FunctorImpl` `class` 派生,你甚至可以定义特殊的 `Functor` 对象。所以,我们可以定义一个 `Functor` 来处理转型。一旦转型工作由程序库来承担,对称性的实现就很容易了。

让我们定义一个 `FunctorDispatcher`, 其作用是将调用分派至任何 `Functor` 对象。这个 `dispatcher` 将内含一个 `BasicDispatcher`, 用来存储 `Functor` 对象。

```
template <class BaseLhs, class BaseRhs = BaseLhs,
          typename ResultType = void>
class FunctorDispatcher
{
    typedef Functor<ResultType,
                    TYPELIST_2(BaseLhs&, BaseRhs&)>
        FunctorType;
    typedef BasicDispatcher<BaseLhs, BaseRhs, ResultType,
                            FunctorType>
        BackendType;
    BackendType backEnd_;
```

```
public:
    ...
};
```

FunctorDispatcher 将 **BasicDispatcher** 具现体作为其后端 (back end) 使用。**BasicDispatcher** 存储着 **FunctorType** 对象, 那是一种 **Functor**, 接受两个参数 (**BaseLhs** 和 **BaseRhs**), 传回 **ResultType**。

在 **FunctorDispatcher::Add()** 成员函数中, 经由 **FunctorImpl** class 派生, 我们定义了一个特殊的 **FunctorImpl** class。这个特殊的 class (名为 **Adapter**, 如后所示) 负责将引数转换为正确型别; 换言之, 它负责将引数型别从 **BaseLhs** 和 **BaseRhs** 转化为 **SomeLhs** 和 **SomeRhs**。

```
template <class BaseLhs, class BaseRhs = BaseLhs,
         ResultType = void>
class FunctorDispatcher
{
    ... as above ...
    template <class SomeLhs, class SomeRhs, class Fun>
    void Add(const Fun& fun)
    {
        typedef
            FunctorImpl<ResultType, TYPELIST_2(BaseLhs&, BaseRhs&)>
            FunctorImplType;
        class Adapter : public FunctorImplType
        {
            Fun fun_;
            virtual ResultType operator()(BaseLhs& lhs, BaseRhs& rhs)
            {
                return fun_(
                    dynamic_cast<SomeLhs&>(lhs),
                    dynamic_cast<SomeRhs&>(rhs));
            }
            virtual FunctorImplType* Clone() const
            { return new Adapter; }
        public:
            Adapter(const Fun& fun) : fun_(fun) {}
    };
    backEnd_.Add<SomeLhs, SomeRhs>(
        FunctorType((FunctorImplType*)new Adapter(fun));
    )
};
```

Adapter class 完成的工作和 **trampoline function** 完全一样。由于仿函数拥有状态, 所以 **Adapter** 聚集 (aggregate) 了一个 **Fun** 对象——这对一个 **trampoline function** 来说是不可能的。成员函数 **Clone()** 则是 **Functor** 所需要的, 其语义不言自明。

FunctorDispatcher::Add 具有极广泛的用途。经由它你不仅可以注册函数指针, 还可以注册几乎任何你想要的仿函数——甚至泛化仿函数。**Add()** 中的 **Fun** 型别的唯一条件是, 必须接受 **function call** 操作符, 其引数型别为 **SomeLhs** 和 **SomeRhs**, 返回型别可转换为 **ResultType**。下面的例子中, 我将两个不同的仿函数注册到 **FunctorDispatcher** 对象中。

```

typedef FunctorDispatcher<Shape> Dispatcher;
struct HatchRectanglePoly
{
    void operator()(Rectangle& r, Poly& p) {
        ...
    }
};
struct HatchEllipseRectangle
{
    void operator()(Ellipse& e, Rectangle& r) {
        ...
    }
};
...
Dispatcher disp;
disp.Add<Rectangle, Poly>(HatchRectanglePoly());
disp.Add<Ellipse, Rectangle>(HatchEllipseRectangle());

```

这两个仿函数不需要有任何关联（例如继承自一个公共基类等等）。它们需要做的只不过是：为它们打算处理的型别实现 `operator()`。

为 `FunctorDispatcher` 实现对称性，就像在 `FnDispatcher` 中实现对称性一样。`FunctorDispatcher::Add` 定义了一个新的 `ReverseAdapter` 对象，用来执行转型任务，并颠倒调用次序。

11.9 引数的转型: `static_cast` 或 `dynamic_cast`?

前面所有代码中，转型都是通过安全的 `dynamic_cast` 来完成，但 `dynamic_cast` 的安全是以执行期效率的损失为代价。

注册时期你就已经知道，你的函数或仿函数会为一对“明确”而“已知”的型别触发。当 `double dispatcher` 经由其所实现的机制在 `map` 中找到一个目标时，它是知道实际型别的。那么让 `dynamic_cast` 再次检查正确性似乎是一种浪费，因为简单的 `static_cast` 可以达到相同结果，需要的时间却少得多。

但是在两种情况下，`static_cast` 会失败，只能依靠 `dynamic_cast` 进行转换。第一种情况出现在虚继承（`virtual inheritance`）。请看以下 `class` 继承体系：

```

class Shape { ... };
class Rectangle : virtual public Shape { ... };
class RoundedShape : virtual public Shape { ... };
class RoundedRectangle : public Rectangle,
    public RoundedShape { ... };

```

图 11.2 以图形表示这个继承体系中的 `classes` 之间的关系。

这或许不是一个灵巧的 `class` 继承体系，但是设计 `class libraries` 时你必须面对这样一个事实：你永远不知道客户想做什么。尽管使用钻石形 `class` 继承体系有很多禁忌，但在某些合情合理的情

况下它还是必要的，所以我们定义的 double dispatchers 也必须适用于钻石形 class 继承体系。

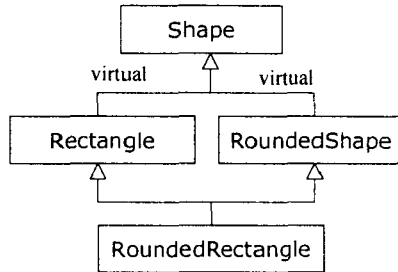


图 11.2 使用了虚继承的钻石形 class 继承体系

实际上，直至今日，这个 dispatcher 运作得很好。但如果将 `dynamic_casts` 替换为 `static_casts`，那么只要你想将 `Shape&` 转换为 `Rectangle&`、`RoundedShape&`、`RoundedRectangle&` 中的任何一个，你都会得到编译错误，原因是虚继承和普通继承的工作方式大不相同。虚继承提供了一种方法，可以让多个 derived classes 共享同一个 base class 对象。在为派生对象分布内存时，编译器不能简单地将 base object 和 derived class 所添加的东西混在一起。

在某些多重继承实作品中，每一个 derived object 都保存着一个“指向 base object”的指针。一旦将 derived 转换为 base 时，编译器会使用那个指针。但是 base object 并没有保存“指向 derived object”的指针。从实践的观点来看，这意味着将 derived type 对象转换为 virtual base type 对象后，任何编译期机制都无法再次找回那个 derived object。通过 `static_cast`，你无法将一个 virtual base 对象转换为一个 derived type 对象。（译注：建议参考《*Inside the C++ Object Model*》by Lippman，中译本《深度探索 C++ 对象模型》）

但是 `dynamic_cast` 运用更高级手段来获知 classes 间的关系，而且即使在 virtual bases 出现的情况下，它也会工作得很好。简而言之，如果在 class 继承体系中使用了虚继承，就必须使用 `dynamic_cast`。

第二，让我们来分析另一种情况。这种情况下我们有一个类似的 class 继承体系，但未使用虚继承，只使用了一般的多重继承。

```

class Shape { ... };
class Rectangle : public Shape { ... };
class RoundedShape : public Shape { ... };
class RoundedRectangle : public Rectangle,
    public RoundedShape { ... };
  
```

图 11.3 显示其继承图。

这个 class 继承体系的形状和先前一样，但对象的结构大不相同。现在，`RoundedRectangle` 有两个截然不同的 `Shape` 子对象。这意味着如今从 `RoundedRectangle` 到 `Shape` 的转换存在歧义

(模棱两可): 你指的是哪一个 Shape? 是 RoundedShape 中的 Shape, 还是 Rectangle 中的 Shape? 同理, 你甚至无法将 Shape& 静态转换为 RoundedRectangle&, 因为编译器不知道该选取哪一个 Shape 子对象。

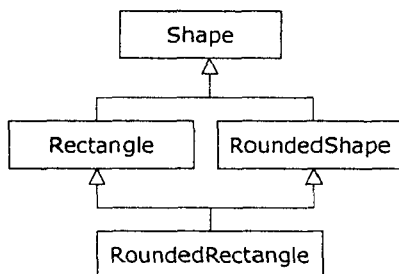


图 11.3 使用非虚继承的钻石形 class 继承体系

我们再次遇上了麻烦。请看下面的代码:

```

RoundedRectangle roundRect;
Rectangle& rect = roundRect; // Unambiguous implicit conversion
Shape& shape1 = rect;
RoundedShape& roundShape = roundRect; // Unambiguous implicit
// conversion
Shape& shape2 = roundShape;
SomeDispatcher d;
Shape& someOtherShape = ...;
d.Go(shape1, someOtherShape);
d.Go(shape2, someOtherShape);

```

这里很重要的一点是: 将 Shape& 转换为 RoundedShape& 时, dispatcher 用的是 dynamic_cast。如果想注册一个 trampoline function 用来将 Shape& 转换为 RoundedRectangle&, 那么由于存在歧义, 将会出现编译错误。

如果 dispatcher 使用的是 dynamic_cast, 就不会有任何问题。对于 RoundedRectangle 中的任何一个 Shape 子对象, dynamic_cast<RoundedRectangle&> 都适用, 并且可以产生出正确的对象。可见除了动态转换, 我们别无选择。dynamic_cast 操作符就是用来获得 class 继承体系中的正确对象——无论这个继承体系的结构有多么复杂。

综合以上分析, 我们的结论是: 在一个 class 继承体系中, 如果多处出现同一个 base class (无论是否使用虚继承), 你都不能在 dispatcher 中使用 static_cast。

这会带给你一种强烈诱惑: 在所有 dispatchers 中都使用 dynamic_cast, 但我必须补充两点。

- 现实世界中，极少 class 继承体系采用钻石形继承图。这种继承体系非常复杂：它们的缺点往往超过优点。大多数设计者之所以尽量避免使用它，原因就在这里。
- `dynamic_cast` 的速度比 `static_cast` 慢得多。强大功能需要付出成本。有很多用户的 class 继承体系很简单，但需要高速度。如果 `double dispatcher` 使用 `dynamic_cast`，这些用户就只有两种选择：要么白手起家重新实现整个 `dispatcher`，要么不辞辛苦地修改程序库。

Loki 采用的方案是：将“转型”视为一个 policy——`CastingPolicy`（请阅读第 1 章有关 policies 的论述）。这里的 policy 是个 class template，有两个参数：来源型别和目标型别。这个 policy 只显露一个函数：`Cast` 静态函数。以下是 `DynamicCaster` policy class。

```
template <class To, class From>
struct DynamicCaster
{
    static To& Cast(From& obj)
    {
        return dynamic_cast<To&>(obj);
    }
};
```

按照第 1 章介绍的原则，`FnDispatcher` 和 `FunctorDispatcher` 这两个 `dispatchers` 可以使用 `CastingPolicy`。下面是修改后的 `FunctorDispatcher` class，修改之处以粗体表现。

```
template
<
    class BaseLhs,
    class BaseRhs = BaseLhs,
    ResultType = void,
    template <class, class> class CastingPolicy = DynamicCaster
>
class FunctorDispatcher
{
    ...
    template <class SomeLhs, class SomeRhs, class Fun>
    void Add(const Fun& fun)
    {
        class Adapter : public FunctorType::Impl
        {
            Fun fun_;
            virtual ResultType operator()(BaseLhs& lhs,
                BaseRhs& rhs)
            {
                return fun_(
                    CastingPolicy<SomeLhs, BaseLhs>::Cast(lhs),
                    CastingPolicy<SomeRhs, BaseRhs>::Cast(rhs));
            }
            ... as before ...
        };
    };
};
```

```

        backEnd_.Add<SomeLhs, SomeRhs>(
            FunctorType(new Adapter(fun));
    }
};

```

casting policy 缺省为 `DynamicCaster`。

最后，通过 casting policies，你还可以做一件很有趣的事情。请看图 11.4 的继承体系。其中有两类转型。一类转型不涉及钻石结构，因此，`static_cast` 是安全的。也就是说将 `Shape&` 转型至 `Triangle&` 时，你可以使用 `static_cast`。但若将 `Shape&` 转型为 `Rectangle&` 或其任何派生类，就不能使用 `static_cast`，必须改用 `dynamic_cast`。

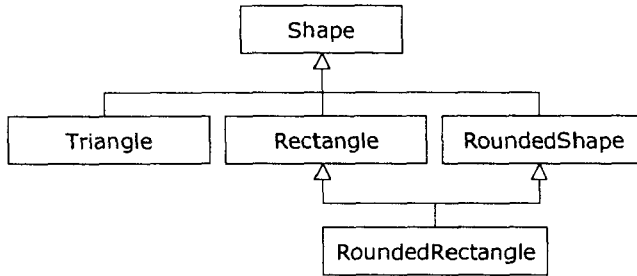


图 11.4 拥有局部钻石形状的 class 继承体系

假设你想为这个 class 继承体系定义自己的转型策略 (casting policy)，例如 `ShapeCast`。你可以将它缺省为 `dynamic_cast`。然后针对特殊情况特化之：

```

template <class To, class From>
struct ShapeCaster
{
    static To& Cast(From& obj)
    {
        return dynamic_cast<To&>(obj);
    }
};

template<>
class ShapeCaster<Triangle, Shape>
{
    static Triangle& Cast(Shape& obj)
    {
        return static_cast<Triangle&>(obj);
    }
};

```

现在，你做到了两全其美——任何时间你都拥有高速转型，必要时则拥有安全转型。

11.10 常数时间的 Multimethods: 原始速度 (Raw Speed)

也许你曾考虑使用 `static dispatcher`, 但发现它耦合性太强; 或者你曾尝试使用 `map-based dispatcher`, 但发现它太慢。你不甘心妥协: 你需要绝对的速度和绝对的可扩充性, 即使为此付出代价也在所不惜。

这种情况下你付出的代价是: 修改你的 `classes`。你得允许 `double dispatcher` 在你的 `classes` 中安插某些机关 (hooks), 以便日后加以利用。

这个机会将为 `double-dispatch` 引擎的实现带来新思路。转型支持依然不变。但“存储”和“获取”处理函数 (handlers) 的方法必须改变——毕竟对数时间不是常数时间。

为找到更好的分派机制, 让我们再次问问自己: 什么是 `double dispatching`? 你可以把它想象为在一个二维空间中寻找一个 `handler` 函数 (或仿函数)。二维空间的坐标轴之一是左操作数的型别, 坐标轴之二是右操作数的型别。在两个型别交叉处, 你可以找到它们各自的 `handler` 函数。图 11.5 表现的是“针对两个 `classes` 继承体系”的 `double dispatch`。这两个体系一是 `Shapes`, 另一是 `DrawingDevices`。此处的 `Handlers` 是一些绘图函数, 它们知道如何在每一个具象的 `DrawingDevice` 对象上产生每一个具象的 `Shape`。

稍做思考你就会想到, 在这个二维空间中, 如果你需要常数时间的查找, 就必须借助于一个二维矩阵 (2-dim matrix) 进行索引访问 (indexed access)。

想法很快就冒了出来。每一个 `class` 必须有个独一无二的整数值, 作为 `dispatcher` 矩阵中的索引。该整数值必须能被每一个 `class` 访问, 而且访问时间为常数。这里, 虚函数会带来帮助。当你发出一个 `double-dispatch call` 时, `dispatcher` 会从两个对象中取得两个索引, 然后取得矩阵中对应的 `handler` 并执行之。成本是: 两个虚调用, 一个矩阵索引操作, 一个通过函数指针的调用。成本固定不变。

这个想法看上去很不错, 但一些细节不容易处理好。例如索引的维护非常麻烦。对于每一个 `class`, 你都必须分配独一无二的整数 ID, 并希望能够在编译期纠举出重复值。这些整数 ID 必须从零开始, 而且无间断——否则会浪费矩阵的存储空间。

更好的方案是, 将索引交给 `dispatcher` 本身去管理。每一个 `class` 都存储一个静态整数型变量; 其值起始为 `-1`, 表示“未被赋值”。准备一个虚函数, 传回该静态变量的 `reference`, 允许 `dispatcher` 在执行期间修改它。当你向矩阵添加新的 `handlers` 时, `dispatcher` 会取用 ID, 如果其值为 `-1`, 就将矩阵中的下一个可用值赋予它。

这项实作的要点是一个简单的宏 (macro): 在你的 `class` 继承体系, 你必须将它添加到每一个 `class` 中。

```
#define IMPLEMENT_INDEXABLE_CLASS(SomeClass)
    static int& GetClassIndexStatic()\
{\
```

```

        static int index = -1;\
        return index;\
    }\
    virtual int& GetClassIndex()\
    {\
        assert(typeid(*this) == typeid(SomeClass));\
        return GetClassIndexStatic();\
    }

```

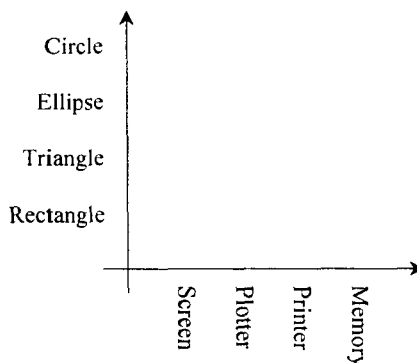


图 11.5 针对 Shapes 和 DrawingDevices 的分派 (dispatching)

如果想让某个 class 支持 multiple dispatch, 就必须在 class 的 public 区添加这个宏⁴¹。

BasicFastDispatcher class template 提供的功能和前面定义的 **BasicDispatcher** 所提供者完全相同, 但使用不同的“存储”和“获取”机制。

```

template
<
    class BaseLhs,
    class BaseRhs = BaseLhs,
    typename ResultType = void,
    typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
>
class BasicFastDispatcher
{
    typedef std::vector<CallbackType> Row;
    typedef std::vector<Row> Matrix;
    Matrix callbacks_;
    int columns_;

```

⁴¹ 没错, 是 multiple dispatch, 而非仅仅 double dispatch。你可以轻易将这个基于索引的方案一般化, 用以支持 multiple dispatch。

```

public:
    BasicFastDispatcher() : columns_(0) {}
    template <class SomeLhs, SomeRhs>
    void Add(CallbackType pFun)
    {
        int& idxLhs = SomeLhs::GetClassIndexStatic();
        if (idxLhs < 0)
        {
            callbacks_.push_back(Row());
            idxLhs = callbacks_.size() - 1;
        }
        else if (callbacks_.size() <= idxLhs)
        {
            callbacks_.resize(idxLhs + 1);
        }
        Row& thisRow = callbacks_[idxLhs];
        int& idxRhs = SomeRhs::GetClassIndexStatic();
        if (idxRhs < 0)
        {
            thisRow.resize(++columns_);
            idxRhs = thisRow.size() - 1;
        }
        else if (thisRow.size() <= idxRhs)
        {
            thisRow.resize(idxRhs + 1);
        }
        thisRow[idxRhs] = pFun;
    }
};

```

其中 **callback** 矩阵是个 **vector**，存储的是“**MappedType** 所构成的 **vectors**”。**FastDispatcher::Add** 函数执行以下一系列动作：

1. 调用 **GetClassIndexStatic**，取得任何 **class** 的 ID。
2. 如果有一个索引未被初始化，或两个索引都未被初始化，就执行初始化和调整工作。对于未初始化的索引，**Add** 会扩展这个矩阵，接纳新增元素。
3. 在矩阵的适当位置插入 **callback**。

成员变量 **columns_** 记录的是迄今为止增加的纵列数。严格来说 **columns_** 是多余的，因为在矩阵中查找最大的横行（**row**）长度，会得到相同结果。但是有了 **columns_** 我们会更方便。

现在，**BasicFastDispatcher::Go** 很容易实现了。主要的不同是，**Go** 使用了虚函数 **GetClassIndex**。

```

template <...>
class BasicFastDispatcher
{
    ... as above ...
    ResultType Go(BaseLhs& lhs, BaseRhs& rhs)
    {
        int& idxLhs = lhs.GetClassIndex();

```

```

int& idxRhs = rhs.GetClassIndex();
if (idxLhs < 0 || idxRhs < 0 ||
    idxLhs >= callbacks_.size() ||
    idxRhs >= callbacks_[idxLhs].size() ||
    callbacks_[idxLhs][idxRhs] == 0)
{
    ... error handling goes here ...
}
return callbacks_[idxLhs][idxRhs].callback_(lhs, rhs);
}
};

```

让我们回顾一下本节。我们定义了一个 matrix-based dispatcher，只要为每个 class 分配一个整数索引，它便可以在常数时间内访问 callback 对象。此外它还会将其辅助数据（亦即与 classes 相对应的索引）自动初始化。每一个打算使用 BasicFastDispatcher 的 class 内都必须添加一行宏（macro）：IMPLEMENT_INDEXABLE_CLASS(YourClass)。

11.11 将 BasicDispatcher 和 BasicFastDispatcher 当做 Policies

在速度方面，matrix-based BasicFastDispatcher 比 map-based BasicDispatcher 更出色。但是，巧妙的高阶 classes (FnDispatcher 和 FunctorDispatcher) 都是根据 BasicDispatcher 构造而成的。我们是不是应该开发两个新的 classes (FnFastDispatcher 和 FunctorFastDispatcher)，以 BasicFastDispatcher 作为其后端 (back end) 呢？

不，更好的做法是改装 FnDispatcher 和 FunctorDispatcher，让它们根据某个 template 参数决定使用 BasicDispatcher 或 BasicFastDispatcher 为后端。也就是说，让 dispatcher 成为 FnDispatcher 和 FunctorDispatcher 的一个 policy，就像我们处理“转型策略 (cast policy)”那样。

“将 dispatcher 变成一个 policy”是件轻松的任务，因为 BasicDispatcher 和 BasicFastDispatcher 拥有相同的调用接口。这使得二者之间的替换就像改变一个 template 引数那样简单。

以下是修改后的 FnDispatcher 声明 (FunctorDispatcher 的声明与此类似)。修改部分以粗体显示。

```

template
<
    class BaseLhs,
    class BaseRhs = BaseLhs,
    typename ResultType = void,
    template <class, class>
        class CastingPolicy = DynamicCaster,
    template <class, class, class, class>
        class DispatcherBackend = BasicDispatcher
>
class FnDispatcher; // similarly for FunctorDispatcher

```

至于两个 classes 本身只作极少修改。

表 11.1 DispatcherBackend Policy 的条件

表达式 (expression)	返回型别	注释
拷贝, 赋值, 交换, 摧毁 (copy, assign, swap, destroy)		value 语义
backEnd.Add<SomeLhs, SomeRhs>(callback)	void	针对 SomeLhs 和 SomeRhs 型别, 向 backEnd 对象添加一个 callback
backEnd.Go(BaseLhs&, BaseRhs&)	ResultType	为两个对象执行查找和分派操作。如果未找到 handler, 就抛出 std::runtime_error 异常
backEnd.Remove<SomeLhs, SomeRhs>()	bool	针对 SomeLhs 和 SomeRhs 型别, 删除 callback。如果有 callback, 传回 true
backEnd.HandlerExists<Some Lhs, SomeRhs>()	bool	如果已为 SomeLhs 和 SomeRhs 型别注册 callback, 就传回 true。此举并不添加 callback

让我们阐述 DispatcherBackEnd policy 的条件。首先, 很明显, 它必须是个 template, 并带四个参数。参数语义依次为:

- 左操作数型别
- 右操作数型别
- callback 的返回型别
- callback 的型别

表 11.1 中, BackendType 表示 dispatcher 后端 (back end) template 具现体, backEnd 表示该型别之变量。表中有一些没提过的函数——别担心。完整的 dispatcher 必定会带来其他一些函数, 用来删除 callback, 在不调用 callback 的情况下执行“被动”查找, 等等。这些都是实现细节; 你可以在 Loki 源码中的 MultiMethods.h 文件内看到它们。

11.12 展望

泛化 (generalization) 就在眼前。我们可以提取有关 double dispatch 的研究成果, 利用它们来实现真正泛化而通用的 multiple dispatch。

这真是太容易了。本章定义了三种 double dispatchers:

- static dispatcher, 由两个 typelists 驱动
- map-based dispatcher, 由一个 map 驱动, 其中 key 是一对 `std::type_info` 对象⁴²
- matrix-based dispatcher, 由一个 matrix 驱动, 以独一无二的数值型 class IDs 为索引

我们可以轻易将这些 dispatchers 一般化, 就像下面这样。你可以将 static dispatchers 一般化, 让它们由一个“typelists 形成的 typelist”驱动, 而不是由两个 typelists 驱动。是的, 你可以定义一个由 typelists 组成的 typelist, 因为任何 typelist 也都是型别。下面的 typedef 就定义了一个“由三个 typelists”组成的 typelist, 在 triple-dispatch 情况下, 这三个 typelists 是可能的参与者。最引人注目的是, 最终的 typelist 确实很容易读取。

```
typedef TYPELIST_3
(
    TYPELIST_3(Shape, Rectangle, Ellipse),
    TYPELIST_3(Screen, Printer, Plotter),
    TYPELIST_3(File, Socket, Memory)
)
ListOfLists;
```

你也可以将 map-based dispatcher 一般化, 使它的 key 成为一个“由 `std::type_info` 对象 (而非 `std::pair`) 组成的 vector”。vector 的大小是 multiple-dispatch 所涉及的对象数目。泛化的 BasicDispatcher 大致像下面这样:

```
template
<
    class ListOfTypes,
    typename ResultType,
    typename CallbackType
>
class GeneralBasicDispatcher;
```

其中 template 参数 ListOfTypes 是个 typelist, 内含 multiple dispatch 涉及的基础型别。例如先前那个“在两形状相交区内绘制影线”的例子中, 我们会使用 TYPELIST_2(Shape, Shape)。

藉由多维 array, 我们可以将 matrix-based dispatcher 一般化。你可以通过递归 class template 来构造多维 array。在先前那个“为型别分配数值型 ID”的方案中, 我们就是这么做的。这有着很好的效果: 如果你为了支持 double dispatch 而对继承体系作了一次修改, 那么以后就不必为了支持 multiple dispatch 再次修改它。

所有这些可能的扩充工作只需要普通工作量就可处理好各个细节。multiple dispatch 和 C++ 存在的一个特别糟糕的问题是: 没有统一的办法可以表示“引数数量不定”的函数。

截至目前, Loki 只实现了 double dispatch。至于上面提出的那些有趣的泛化工作, 则是以讨厌的练习形式留给了…唔, 你知道留给了谁☹。

⁴² 被包装为 `OrderedTypeInfo`, 用以简化“比较”和“拷贝”动作。

11.13 摘要

Multimethods 是广义上的虚函数。但是, C++ 执行期基于“单一”class 来分派虚函数, multimethods 则是根据多个 classes 而分派。这样你就可以一次性地为成组（而非单一）的型别实现虚函数。

从其本性来看, Multimethods 最好被实现为一种“语言特性”。遗憾的是 C++ 缺少这样一种特性, 但是我们有不少办法在程序库中实现它。

在应用程序中, 如果需要根据两个（或更多）对象的型别来决定调用哪个算法, 我们就需要使用 multimethods。典型的例子包括: 多态对象之间的碰撞、相交、在各种目标设备上显示对象, 等等。

本章的讨论仅限于“基于两个对象”的 multimethods。负责确定“该调用哪一个合适函数”的对象, 我们称为 double dispatcher。本章总共讨论了以下数种 dispatchers:

- *brute-force* (暴力式) *dispatcher*。这类 dispatcher 必须仰赖（以 typelist 形式提供的）静态型别信息, 并执行线性、非往复式查找, 以找到正确型别。一旦找到目标型别, dispatcher 会在一个 handler 对象中调用一个重载的 (overloaded) 成员函数。
- *map-based dispatcher*。使用一个 map, 其中的 key (键值) 为 `std::type_info` 对象, value (实值) 为 callback (一个函数指针或仿函数)。采用二分查找法来寻找型别。
- *constant-time* (常数时间) *dispatcher*。这是速度最快的 dispatcher, 但你必须修改相应的 classes。需要的修改是: 在每一个打算使用 constant-time dispatcher 的 class 中添加一个宏。每次分派 (dispatch) 的成本是两次虚调用、两次数值测试、一次矩阵 (matrix) 元素访问。

在后两者身上, 我们可以实现以下更高级的功能:

- 自动化转换 (*automated conversions*) ——可别和 C++ 的自动转换 (*automatic conversions*) 弄混了。由于上面提到的 dispatchers 缺乏一致性, 所以它们需要客户将对象从基础型别转至派生型别。我们可以提供一个转型层 (casting layer), 通过一个 trampoline function 来处理这些转换。
- 对称性 (*symmetry*)。某些 double-dispatch 的应用天生就是对称的: 在操作的左右两侧, 基于相同的基础型别执行分派动作, 不管元素的左右次序。例如在一个“碰撞检测”器中, “飞船撞击鱼雷”和“鱼雷撞击飞船”的结果是一样的。如果在程序库中提供“对称性”支持, 用户的代码就会更小, 出现错误的几率也会降低。

brute-force dispatcher 直接支持这些高级特性。之所以能够如此, 是因为它拥有丰富的型别信息。另两种 dispatchers 采用不同的方法, 它们增加了一个额外层, 用以实现自动化转换 (*automated conversions*) 和对称性。实现这个额外层时, “函数”版本的 double dispatchers 与“仿函数”版本并不相同, 而且更有效率。

表 11.2 就本章定义的三种 dispatcher 进行了比较。可以看到, 它们都不完美。你应该针对具体

情况，选择最符合你需要的解决方案。

表 11.2 各种 double dispatch 实作法的比较

	Static Dispatcher (StaticDispatcher)	Logarithmic Dispatcher (BasicDispatcher)	Constant-Time Dispatcher (BasicFastDispatcher)
classes 少时的速度	很快	中等	快
classes 多时的速度	慢	快	很快
带来的依存性	强	弱	弱
需要修改现有 classes	否	否	为每个 class 添加一个宏
编译期安全性	很高	高	高
执行期安全性	很高	高	高

11.14 Double Dispatcher 要点概览

- Loki 定义了三种基本的 double dispatchers: StaticDispatcher、BasicDispatcher、BasicFastDispatcher。
- StaticDispatcher 声明如下：

```
template
<
    class Executor,
    bool symetric,
    class BaseLhs,
    class TypesLhs,
    class BaseRhs = BaseLhs,
    class TypesRhs = TypesLhs,
    typename ResultType = void
>
class StaticDispatcher;
```

其中：

- BaseLhs 是左侧基础型别（base left-hand type）。
- TypesLhs 是个 typelist，内含 double dispatch 左侧（left-hand side）涉及的那组具体型别。
- BaseRhs 是右侧基础型别（right-hand type）。
- TypesRhs 是个 typelist，内含 double dispatch 右侧（right-hand side）涉及的那组具体型别。

Executor 是个 class, 它提供的函数会在型别被找到后被调用。针对 **TypesLhs** 和 **TypesRhs** 中的每一个型别组合, **Executor** 都必须提供一个重载成员函数 **Fire()**。

ResultType 是重载函数 **Executor::Fire()** 的返回型别。返回值会被当做 **StaticDispatcher::Go** 的执行结果。

- **Executor** 必须提供一个 **OnError(BaseLhs&, BaseRhs&)** 成员函数, 用以执行错误处理。当 **StaticDispatcher** 遇到一个未知型别, 便会调用 **Executor::OnError()**。
- 下面是个运用实例 (其中 **Rectangle** 和 **Ellipse** 继承自 **Shape**, **Printer** 和 **Screen** 继承自 **OutputDevice**) :

```
struct Painter
{
    bool Fire(Rectangle&, Printer&);
    bool Fire(Ellipse&, Printer&);
    bool Fire(Rectangle&, Screen&);
    bool Fire(Ellipse&, Screen&);
    bool OnError(Shape&, OutputDevice&);
};

typedef StaticDispatcher
<
    Painter,
    Shape,
    TYPELIST_2(Rectangle, Ellipse),
    OutputDevice,
    TYPELIST_2(Printer&, Screen),
    bool
>
Dispatcher;
```

- **StaticDispatcher** 必须实作出成员函数 **Go()**, 该函数接受一个 **BaseLhs&**、一个 **BaseRhs&** 和一个 **Executor&**, 执行分派操作 (**dispatch**)。下面是用法 (延续上述定义) :

```
Dispatcher disp;
Shape* pSh = ...;
OutputDevice* pDev = ...;
bool result = disp.Go(*pSh, *pDev, Painter());
```

- **BasicDispatcher** 和 **BasicFastDispatcher** 实现动态分派, 允许用户在执行期添加 handler functions。
- **BasicDispatcher** 保证在对数时间内找到一个 handler。**BasicFastDispatcher** 保证在常数时间内找到一个 handler——但用户必须修改所有“被分派的 classes”的定义内容。
- 两个 classes 都实现了相同接口, 以下示范 **BasicDispatcher** 的用法:

```
template
<
    class BaseLhs,
    class BaseRhs = BaseLhs,
    typename ResultType = void,
```

```

    typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
>
    class BasicDispatcher;

```

其中:

CallbackType 是处理“分派操作”之对象所属的型别。

BasicDispatcher 和 **BasicFastDispatcher** 存储并调用上述型别之对象。

其他所有参数的含义和 **StaticDispatcher** 之中雷同。

- 这两个 dispatchers 实现出表 11.1 所示函数。
- 除了三种基本 dispatchers 之外, Loki 还定义了两个高级版本: **FnDispatcher** 和 **FunctorDispatcher**。它们将 **BasicDispatcher** 或 **BasicFastDispatcher** 当做 policy 来使用。
- **FnDispatcher** 和 **FunctorDispatcher** 的声明很类似, 如下所示:

```

template
<
    class BaseLhs,
    class BaseRhs = BaseLhs,
    ResultType = void,
    template <class To, class From>
        class CastingPolicy = DynamicCast
    template <class, class, class, class>
        class DispatcherBackend = BasicDispatcher
>
    class FnDispatcher;

```

其中:

BaseLhs 和 **BaseRhs** 是 double dispatch 之中两个继承体系的 base classes。

ResultType 是 callbacks 和 dispatcher 的返回型别。

CastingPolicy 是带有两个参数的 class template。它必须实现一个静态成员函数 **Cast()**, 接受一个 reference 指向 **From**, 传回一个 reference 指向 **To**。**DynamicCaster** 和 **StaticCaster** 分别缺省使用 **dynamic_cast** 和 **static_cast**。

DispatcherBackend 是个 class template, 它所实现的接口和 **BasicDispatcher**、**BasicFastDispatcher** 一样, 如表 11.1 所示。

- **FnDispatcher** 和 **FunctorDispatcher** 都提供成员函数 **Add()**, 或是它们的基础 handler 类型。对 **FnDispatcher** 来说, 其基础 handler 类型是 **ResultType (*)(BaseLhs&, BaseRhs&)**。对 **FunctorDispatcher** 来说, 其基础 handler 类型是 **Functor<ResultType, TYPELIST_2(BaseLhs&, BaseRhs&)>**。请参阅第 5 章关于 **Functor** 的介绍。
- **FnDispatcher** 还提供一个 template 函数, 用于在引擎中注册 callbacks:

```

void Add<SomeLhs, SomeRhs,
    ResultType (*)(SomeLhs&, SomeRhs&),
    bool symmetric>();

```

- 如果通过先前代码所演示的 `Add()` 成员函数注册了 `handlers`，你将获得“自动化转型（automated casting）”和“可选之对称性（optional symmetry）”等好处。
- `FunctorDispatcher` 提供一个 `template` 成员函数 `Add()`：

```
template <class SomeLhs, class SomeRhs, class F>
void Add(const F& fun);
```
- 上述的 `F` 可以是 `Functor` 对象（见第 5 章）所能接受的任何型别，甚至是另一个 `Functor` 具现体。`F` 对象必须支持 `function call` 操作符，此操作符的引数型别为 `BaseLhs&` 和 `BaseRhs&`，返回型别必须可以转换为 `ResultType`。
- 如果没有找到任何 `handler`，所有分派引擎（`dispatch engine`）都会丢出一个型别为 `std::runtime_error` 的异常。

附录

一个超迷你的多线程程序库

A Minimalist Multithreading Library

任何时刻，一个多线程程序（multithreaded program）有多个执行点（points of execution）。这意味着在多线程程序中你可以同时执行多个函数。在多处理器（multiprocessor）计算机中，不同的线程真的会同时运行。在单处理器（single-processor）计算机中，如果操作系统支持多线程，它会运用 **time slicing**（分时）技术将每个线程切割为较短的时间间隔、暂停执行、给予另一个线程一些处理器时间（processor time）。多线程带给用户“多件事情同时发生”的感觉，例如在一个字处理程序中，程序在等待用户输入文字的同时可以检查语法。

用户不喜欢见到沙漏光标；所以，身为程序员，我们必须编写多线程程序。不幸的是，多线程虽然带给用户快乐，却往往带给程序员困难，甚至造成程序难以调试。此外，多线程议题遍及应用程序设计范畴。要想让一个程序库在多线程环境下也能安全工作，在程序库外部是无法做到的；它必须构建于内部——即使程序库本身不使用线程。

因此，本书提供的各个组件不能忽略线程问题（喔，是的，它们当然可以忽略；但如此一来一旦面对多线程，它们之中的大多数会毫无用处）。现代应用程序愈来愈多通过多线程执行；所以，如果出于懒惰的因素而忽略多线程，将会令人遗憾。

本附录提供了一些工具和技术，为撰写可移植的、多线程的 C++ 面向对象程序奠定坚实基础。本附录没有对多线程编程作详尽介绍——这是一个迷人但巨大的领域。想在本书附带介绍一个完整的多线程程序库（threading library），注定徒劳无功。我们的目标是设计一个最小化多线程程序库，帮助我们撰写多线程组件。

与现代操作系统提供的众多功能相比，Loki 的多线程能力很弱。因为它的目标只是提供多线程环境下的安全组件。然而从正面来说，本附录定义的同步（synchronization）概念比传统的 **mutexes**（互斥体）和 **semaphores**（信号量）具有更高层次，有助于设计任何面向对象多线程程序。

A.1 多线程的反思

在多处理器计算机上，多线程的优点很明显。但在单处理器计算机上，多线程就显得有点可笑。为什么要用“分时”算法降低处理器的速度呢？很明显，你不会得到任何本质上的好处。没有奇迹发生——还是只有一个处理器；所以从整体来看，多线程实际上会稍微降低效率，因为它要完成额外的交换 (swapping) 和簿记 (bookkeeping) 工作。

但即使在单处理器计算机上，多线程也很重要；原因在于资源 (resource) 的使用效率。在一台典型的现代计算机中，资源比处理器多得多。你有磁盘驱动器、调制解调器 (modem)、网络卡、打印机等设备。这些资源实体上是独立的，因而可以同时工作。当磁盘正在旋转、打印机正在打印的时候，处理器没有理由不工作——但如果你的应用程序和操作系统采用独占式的单线程执行模式，那种情况就会发生。当通过 modem 传输数据时，如果你的程序不允许你做任何其他事情，你一定很不高兴。

同样地，即使是处理器，也有可能在某一段时间内不被使用。当你编辑一个 3D 图像的时候，鼠标“移动”与“点击”之间会有一段短暂的时间间隔，在处理器看来，那是一段漫长光阴。如果绘图程序可以利用那些空闲时间做些有用的事情，例如光线追踪 (ray tracing)、隐线处理 (hidden lines) 等等，一定很不错。

多线程的主要替代方案是异步执行 (asynchronous execution)，它提供一种 callback (回调) 模型：启动一个操作，并注册一个函数，一旦操作完成即调用该函数。和多线程相比，异步执行的主要缺点在于：它带来的是“多状态 (state-rich)”程序。采用异步执行，从一处到另一处，你无法采用同一算法；你只能存储一个状态值，并让 callbacks 来改变它。就算是在最简单的操作中，维护这种状态值也很麻烦。

真正的线程不存在这样的问题。每一个线程都有一个“由其执行点 (即线程目前正在执行的语句) 给出的隐含状态”。你可以轻易跟随一个线程的行为，因为这就像跟随一个简单函数一样。

“执行点”正是你在异步执行中必须手工管理的东西 (异步编程的主要问题是：我现在在哪儿？)。总之，多线程程序可以采用同步执行 (synchronous execution) 模型，这是它好的方面。

但是，线程一旦开始共享资源 (例如内存内的数据)，就会暴露出很大问题。因为任何时候线程都可以被其他线程中断 (是的，“任何”时候，包括正对一个变量赋值的时候)，所以你所以为的原子型操作 (atomic operations) 其实并非不可切割。如果对一块数据进行无组织的多线程访问，一定会造成致命伤害。

在单线程编程之中，数据的健全性通常是在函数的入口和出口获得保证。例如 String class 的赋值运算 (operator=) 认为，在入口和出口，String 对象一定是有效的 (valid)。然而在多线程编程中你必须保证，即使在赋值操作“期间”，String 对象也必须是有效的，因为另一个线程随时有可能中断赋值操作，并对 String 对象执行另一个操作。尽管在单线程编程中你已习惯将函数视为不可切割的所谓原子型操作 (atomic operations)，但在多线程编程中，你必须

明确表明哪些是原子操作。总之，在资源共享方面，多线程程序存在很大的麻烦，这是它不好的一面。

大多数多线程编程技术的焦点都放在如何提供同步对象 (*synchronization objects*)，从而让你可以串行访问 (*serialize access*) 共享资源。如果你的操作不想被切割，你就得锁定一个同步对象。其他线程如果想锁定同一个同步对象，它们就会被高挂扣住 (*put on hold*)。你修改数据（并让它处于某种一致状态），而后解开同步对象。此时其他线程便可锁定那个同步对象，并获得对数据的访问权。通过这一有效方法，每一个线程都在一致的数据上工作。

以下章节定义了多种锁件 (*locking objects*)。这里提供的同步对象并不全面，但藉由它们你还是可以完成大量的多线程编程任务。

A.2 Loki 的做法

为了处理多线程问题，Loki 定义了 *ThreadingModel* policy。那是一个 *template*，只接受一个引数。该引数是一个“希望拥有多线程便利性”的 C++ 型别：

```
template <typename T>
class SomeThreadingModel // 译注：Loki 之中并没有这样一个 class。这只是个代名。
{
    ...
};
```

下面章节中，我们将在概念和功能上对 *ThreadingModel* 逐步充实。Loki 只定义了一个线程模型，大多数 Loki 组件都以它为缺省模型。

A.3 整数型别上的原子操作 (Atomic Operations)

假设 *x* 是个 *int* 变量，考虑下面语句：

```
++x;
```

在一本专论设计的书中分析一个简单的递增语句，似乎很滑稽；但这正是多线程中存在的事实——小问题影响大设计。

为了累加 *x*，处理器必须完成三个动作：

1. 从内存中得到变量。
2. 在处理器的算术逻辑单元 (ALU, *arithmetic logic unit*) 中递增这个变量。ALU 是可以执行递增运算的唯一地方；内存本身并无算术运算能力。
3. 将变量写回内存。

第一个动作读 (**Read**) 数据，第二个动作修改 (**Modify**) 数据，第三个动作写 (**Write**) 数据，因而这三步骤被称作 *read-modify-write* (**RMW**) 操作。

现在，假设递增操作发生于多处理器架构中。为了获得最大效益，在 RMW 操作的“修改”期间，处理器开放内存总线 (*memory bus*)。这样一来，当第一个处理器对变量执行递增运算时，

另一个处理器也可以访问内存，从而更良好地运用资源。

不幸的是另一个处理器也可能对同一个整数执行 RMW 操作。假设 x 初值为 0，有两个递增运算作用于 x 身上，分别由处理器 P1 和 P2 按以下次序执行：

1. P1 锁定内存总线并取得 x 。
2. P1 释放内存总线。
3. P2 锁定内存总线并取得 x （目前为 0）。在此同时，P1 于其 ALU 中递增 x ，结果为 1。
4. P2 释放内存总线。
5. P1 锁定内存总线并将 1 写入 x 。在此同时，P2 于其 ALU 中递增 x 。由于 P2 取得的值是 0，所以结果也是 1。
6. P1 释放内存总线。
7. P2 锁定内存总线并将 1 写入 x 。
8. P2 释放内存总线。

最终结果是，虽然有两个递增运算发生在 x 身上，且 x 初值为 0，但最后结果却是 1。这是一个错误结果。更糟糕的是，两个处理器（线程）都无法判断递增运算失败，因而不会重新执行此运算。在多线程世界中，任何东西都不具“原子性（不可切割性）”——即使简单如整数递增运算也不例外。

有些方法可使递增运算变得具有原子性。最有效益的方法是利用处理器本身能力。某些处理器提供总线（bus）锁定操作——RMW 操作还是像前面描述那样发生，但在整个操作过程中，内存总线是锁定的。于是如果 P2 要从内存中取得 x ，得等到 P1 对 x 的递增操作完成之后。这种底层功能通常由操作系统包装在“提供原子递增和递减运算”的 C 函数中。

如果 OS 定义有原子操作，那么，对那些“内存总线宽度”的整数型别（通常是 `int`），OS 通常都会如此这般地处理。Loki 的线程子系统（`Threads.h` 文件）在每一个 `ThreadingModel` 实作品中都定义有 `IntType` 型别。

`ThreadingModel` 中的原子操作基本元素（primitives）大致如下：

```
template <typename T>
class SomeThreadingModel    // 译注：Loki 之中并没有这样一个 class。这只是个代名。
{
public:
    typedef int IntType; // or another type as dictated by the platform
    static IntType AtomicAdd(volatile IntType& lval, IntType val);
    static IntType AtomicSubtract(volatile IntType& lval, IntType val);
    ... similar definitions for AtomicMultiply, AtomicDivide,
    ... AtomicIncrement, AtomicDecrement ...
    static void AtomicAssign(volatile IntType& lval, IntType val);
    static void AtomicAssign(IntType& lval, volatile IntType & val);
};
```

这些基本元素以“待改之值”作为第一参数（请注意，那是个 `non-const reference`，而且有时

会使用饰词 `volatile`)，以另一个操作数（此对一元操作符而言是空缺的）作为第二参数。每一个基本元素都传回“`volatile` 目标值”副本。使用这些基本元素时，返回值非常有用，因为它可以让你检查操作结果。操作之后，如果你像下面这样检查 `volatile` 值：

```
volatile int counter;
...
SomeThreadingModel<Widget>::AtomicAdd(counter, 5);
if (counter == 10) ...
```

那么在加法操作之后，你的程序不会立即检查 `counter`；因为在“`AtomicAdd` 被调用”和“`if` 语句”之间，另一个线程可能会去修改 `counter`。但通常调用 `AtomicAdd()` 之后你需要立即检查 `counter` 值；这种情况下你可以这么做：

```
if (AtomicAdd(counter, 5) == 10) ...
```

此外，两个 `AtomicAssign` 函数共存是必要的，因为即使拷贝动作也可能不是原子操作。假设你的总线宽度是 32 bits，`long` 有 64 bits，那么拷贝一个 `long` 值就涉及两次内存访问。

A.4 Mutexes (互斥体)

Edgar Dijkstra 已经论证过，如果出现多线程，操作系统的线程调度器 (thread scheduler) 就必须提供某种同步对象。如果没有它们，不可能正确编写多线程程序。

Mutexes (互斥体) 是一种重要的同步对象。有了它，线程可以通过一种井然有序 (ordered) 的方式访问共享资源。本节定义 `mutex` 的概念。Loki 并未直接使用 `mutexes`，而是定义了更高阶的同步手段；藉 `mutexes` 之助，你可以轻易实现那些高阶手段。

Mutex 这个字由 **Mutual Exclusive** 组合而成，表达出 *Mutex* 这个原始对象的功能：允许线程彼此互斥地访问同一块资源。

`mutex` 的基本功能包括 `Acquire` (获得) 和 `Release` (释放)。对每个线程来说，如果它需要独占访问某一资源 (例如共享变量)，就必须获得 `mutex`。一个 `mutex` 只能由一个线程获得。如果某线程获得了 `mutex`，其他所有调用 `Acquire()` 的线程都会被阻塞，进入等待状态 (也就是 `Acquire()` 不返回)。一旦拥有 `mutex` 的线程调用 `Release()`，线程调度器会在“等待 `mutex`”的所有线程中选择一个，并将 `mutex` 的拥有权交给它。

这样造成的显著效果是：`mutexes` 成了一种所谓的串行访问控制器 (access serialization device)：对 `mtx` 对象来说，`mtx.Acquire()` 和 `mtx.Release()` 之间的代码都是原子操作 (不会被其他线程干扰)。对任何其他操作来说，如果想获得 `mtx` 对象，就得等到那些原子操作结束。

所以，针对每一块“意欲在线程之间被共享”的资源，你都应该为它分配一个 `mutex` 对象。要注意的是，“意欲被共享的资源”包括 C++ 对象。这些资源身上的每一个“非原子”操作都必须从“获得 `mutex`”开始，以“释放 `mutex`”结束。你可能执行的“非原子”操作包括“在多线程环境下保证安全 (thread-safe)”的对象的 `non-const` 成员函数。

例如, 假设你有一个 `BankAccount` class, 它提供存款函数 `Deposit()`、`Withdraw()`。这些操作除了对一个 `double` 成员变量进行加减运算外, 还需对交易方面的其他信息作记录。如果 `BankAccount` 会被多线程访问, 这两个操作就必须是原子操作。你可以像下面这样做:

```
class BankAccount
{
public:
    void Deposit(double amount, const char* user)
    {
        mtx_.Acquire();
        ... perform deposit transaction ...
        mtx_.Release();
    }
    void Withdraw(double amount, const char* user)
    {
        mtx_.Acquire();
        ... perform withdrawal transaction ...
        mtx_.Release();
    }
    ...
private:
    Mutex mtx_;
    ...
};
```

你或许已经猜到 (如果你以前不知道的话): 针对每一个 `Acquire()`, 调用相应的 `Release()` 如果失败, 结果会很惨。这就好像你锁定了一个 `mutex`, 然后置之不理, 其他所有想获得这个 `mutex` 的线程都会被永远阻塞。因此上述代码中你必须十分小心地实现 `Deposit()` 和 `Withdraw()`, 处理好“异常”和“过早返回”的情况。

为消除这个问题, 很多 C++ 多线程函数都定义了所谓的 `Lock` 对象, 用来初始化 `mutex`。`Lock` 对象的构造函数调用 `Acquire()`, 析构函数调用 `Release()`。这么一来, 如果是在 `stack` 上分配 `Lock` 对象, 就可以确保 `Acquire()` 和 `Release()` 正确匹配——即使出现异常也没有问题。

考虑到可移植性, `Loki` 并没有定义自己的 `mutexes` ——你很可能已经在使用某个多线程程序库, 其中定义有 `mutexes`, 而重复功能是件很让人难受的事情。`Loki` 使用的是“经由 `mutexes` 实现的高阶锁定语义”。

A.5 面向对象编程中的锁定语义 (Locking Semantics)

同步对象和共享资源紧密相连。在面向对象程序中, 资源就是对象。因此, 在面向对象程序中, 同步对象和应用程序的对象紧密相连。

因此, 每一个共享对象都应该聚合 (aggregate) 一个同步对象, 并在每一个“执行修改动作”的成员函数中锁定它, 就像 `BankAccount` 例中所做那样。要构造出一个“支持多线程”的对象, 这是一种正确作法。这种“为每个对象提供一个同步对象”的结构, 称为 *object-level locking*。

但是有时候，如果为每一个对象存储一个 `mutex`，其空间消耗和额外开销太大。这种情况下“一个 `class` 保留一个 `mutex`”的同步策略可能带来帮助。

想想 `String` `class` 的情况。有时候你需要在 `String` 对象上锁定某一操作。但你不希望每个 `String` 都携带一个 `mutex` 对象，那会使 `String` 太大，还会使 `copy` 动作变得更昂贵。这种情况下你可以使用一个静态 `mutex` 对象，用于所有 `Strings`。这个策略提供的是 *class-level locking*。

Loki 定义了两个 `ThreadingModel` policy 实作品：`ClassLevelLockable` 和 `ObjectLevelLockable`。它们分别包装上述的 *class-level locking* 和 *object-level locking* 语义。以下是它们的大致轮廓。

```
template <typename Host>
class ClassLevelLockable
{
public:
    class Lock
    {
    public:
        Lock();
        Lock(Host& obj);
        ...
    };
    ...
};

template <typename Host>
class ObjectLevelLockable
{
public:
    class Lock
    {
    public:
        Lock(Host& obj);
        ...
    };
    ...
};
```

技术上，`Lock` 使得 `mutex` 保持锁定状态。这两个实作品的区别是：对于 `ObjectLevelLockable<T>::Lock`，如果你不传给它一个 `T` 对象，就无法构造它。原因是 `ObjectLevelLockable` 使用的是 *object-level locking*。

在 `Lock` 这个嵌套类 (nested class) 的整个生命期中，`Lock` 都会将对象（或整个 `class`——在 `ClassLevelLockable` 的情况下）锁定。在应用程序中，你可以继承上述任何一个 `ThreadingModel`，然后直接使用 `Lock` 这个内隐类 (inner class)。例如：

```
class MyClass : public ClassLevelLockable <MyClass>
{
    ...
};
```

具体的锁定策略取决于你派生自哪一份 `ThreadingModel` 实作品。表 A.1 总结了现有的所有实作品。

表 A.1 `ThreadingModel` 实作品

Class Template	语义
<code>SingleThread</code>	完全没有提供多线程策略。 <code>Lock</code> 和 <code>ReadLock</code> 只是两个空的 classes
<code>ObjectLevelLockable</code>	对象级锁定语义。每个对象存储一个 mutex。内隐类 <code>Lock</code> 用来锁定 mutex（也就隐隐锁定了对象）
<code>ClassLevelLockable</code>	类级锁定语义。每个类存储一个 mutex。内隐类 <code>Lock</code> 用来锁定 mutex（也就隐隐锁定了某一型别的所有对象）

你可以像下面例子这样，很方便地定义“同步（synchronized）成员函数”：

```
class BankAccount : public ObjectLevelLockable<BankAccount>
{
public:
    void Deposit(double amount, const char* user)
    {
        Lock lock(*this);
        ... perform deposit transaction ...
    }
    void Withdraw(double amount, const char* user)
    {
        Lock lock(*this);
        ... perform withdrawal transaction ...
    }
    ...
};
```

如果出现“过早返回”或“异常”情况，不再会有任何问题发生：语言的不变性（invariants）保证了 mutex 加锁/解锁操作的正确匹配。

空壳接口（dummy interface）`SingleThreaded` 纯粹是为了让接口有其一致性，也为我们带来语法上的一致性。你可以假设在某种多线程环境下撰写程序，而后只要修改线程模型（threading model），便可轻易变更设计。

第 4 章（小型对象分配）、第 5 章（泛化仿函数）和第 6 章（Singletons 实作技术）均用上了 `ThreadingModel` policy。

A.6 可有可无的（Optional）volatile 饰词

C++ 提供有型别饰词 `volatile`；对于每一个“希望在多线程中被共享”的变量，你都应该以 `volatile` 修饰之。但是在单线程模型中最好不要使用 `volatile`，因为它会阻止编译器执行某些重要优化动作。

这就是 Loki 要定义内隐类 `volatileType` 的原因。在 `SomeThreadingModel<widget>` 中，对 `ClassLevelLockable` 和 `ObjectLevelLockable` 而言，`volatileType` 相当于 `volatile widget`，对 `SingleThreaded` 而言则相当于普通的（无任何饰词的）`widget`。你可以在第 6 章看到 `volatileType` 的运用实例。

A.7 Semaphores, Events 和其他好东西

Loki 对多线程的支持就这些。泛用型多线程程序库应该提供一套更丰富的同步对象和函数，例如 `semaphores`、`events`、`memory barriers` 等等。此外，Loki 之中连“启动一个新线程”的函数都没有，这说明了 Loki 的目标是为了达到 `thread safe`（在多线程环境下安全），而不是为了运用线程。

Loki 的未来版本有可能提供完整的多线程模型。多线程是一个可以大大应用泛型编程技术的领域。这里的竞争很激烈——请看看 ACE（Adaptive Communication Environment）这个伟大且具有极高可移植性的多线程程序库（Schmidt 2000）。

A.8 摘要

C++ *Standard* 中并没有提及线程。但是多线程的同步问题已经渗透到应用程序和程序库的设计之中。问题是各个操作系统所支持的线程模型（`threading models`）都大不相同。因此 Loki 定义出高阶同步机制，它和外部提供的线程模型只牵扯最小联系。

`ThreadingModel policy` 和三个“实现了 `ThreadingModel`”的 `class templates` 共同确立出一个平台，用来构造“支持不同线程模型”的泛型组件。你可以在编译期作出选择，决定支持 *object-level locking* 或 *class-level locking*，或者“完全没有锁定”。

object-level locking 策略为应用程序中的每一个对象分配一个同步对象。*class-level locking* 策略则为每一个 `class` 分配一个同步对象。前者速度较快，后者耗用的资源较少。

`ThreadingModel` 的所有实作品都支持一个统一接口（`uniform interface`）。这使得程序库或客户端程序很容易使用统一语法。你可以调整某个 `class` 对锁定的支持程度，而不需要改变其实作。也正是为了这个目的，Loki 定义了一个什么都没做的“单线程模型”。

参考书目

Bibliography

- Alexandrescu, Andrei. 2000a. Traits: The else-if-then of types. *C++ Report*, April.
- 2000b. On mappings between types and values. *C/C++ Users Journal*, October.
- Austern, Matt. 2000. The standard librarian. *C++ Report*, April.
- Ball, Steve, and John Miller Crawford. 1998. Channels for inter-applet communication. *Doctor Dobb's Journal*, September. Available at <http://www.ddj.com/articles/1998/9809/9809a/9809a.htm>.
- Boost. The Boost C++ Library. <http://www.boost.org>.
- Coplien, James O. 1992. *Advanced C++ Programming Styles and Idioms*. Reading, MA: Addison-Wesley.
- 1995. The column without a name: A curiously recurring template pattern. *C++ Report*, February.
- Czarnecki, Krzysztof, and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Reading, MA: Addison-Wesley.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Järvi, Jaakko. 1999a. *Tuples and Multiple Return Values in C++*. TUCS Technical Report No. 249, March.
- 1999b. The Lambda Library. <http://lambda.cs.utu.fi>
- Knuth, Donald E. 1998. *The Art of Computer Programming*. Vol. 1. Reading, MA: Addison-Wesley.
- Koenig, Andrew, and Barbara Moo. 1996. *Ruminations on C++*. Reading, MA: Addison-Wesley.
- Lippman, Stanley B. 1994. *Inside the C++ Object Model*. Reading, MA: Addison-Wesley.
- Martin, Robert. 1996. Acyclic Visitor. Available at <http://objectmentor.com/publications/acv.pdf>.

- Meyers, Scott. 1996a. *More Effective C++*. Reading, MA: Addison-Wesley.
- 1996b. Refinements to smart pointers. *C++ Report*, November-December.
 - 1998a. *Effective C++*, 2nd ed. Reading, MA: Addison-Wesley.
 - 1998b. Counting objects in C++. *C/C++ Users Journal*, April.
 - 1999. `auto_ptr` update. Available at http://www.awl.com/cseng/titles/0-201-63371-X/auto_ptr.html.
请注意: Colvin/Gibbons 技法并未出现于任何论文之中。Meyers 对 `auto_ptr` 所做的注释是迄今对 Greg Colvin 和 Bill Gibbons 所找出的解法的最精确描述。这个技法乃是利用 `auto_ptr` 来解决函数返回问题。
- Schmidt, D. 1996. Reality check. *C++ Report*, March. Available at <http://www.cs.wustl.edu/~schmidt/editorial-3.html>.
- 2000. The ADAPTIVE Communication Environment (ACE). Available at <http://www.cs.wustl.edu/~schmidt/ACE.html>.
- Stevens, Al. 1998. Undo/Redo redux. *Doctor Dobb's Journal*, November.
- Stroustrup, Bjarne. 1997. *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley.
- 2000. Wrapping calls to member functions. *C++ Report*, June.
- Sutter, Herb. 2000. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Reading, MA: Addison-Wesley.
- Van Horn, Kevin S. 1997. Compile-time assertions in C++. *C/C++ Users Journal*, October. Available at <http://www.xmission.com/~ksvhsoft/ctassert/ctassert.html>.
- Veldhuizen, Todd. 1995. Template metaprograms. *C++ Report*, May. Available at <http://extreme.indiana.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html>.
- Vlissides, John. 1996. To kill a singleton. *C++ Report*, June. Available at <http://www.stat.cmu.edu/~lamj/sigs/c++-report/cppr9606.c.vlissides.html>.
- 1998. *Pattern Hatching*. Reading, MA: Addison-Wesley.
 - 1999. Visitor in frameworks. *C++ Report*, November-December.

索引

Index

A

Abstract Factory design pattern, 69, 40-51
 architectural role of, 219-222
 basic description of, 219-234
 implementing, 226-233
 interface, generic, 223-226
 quick facts, 233-234
AbstractEnemyFactory, 221-222, 224-227
AbstractFactory, 219-234
AbstractProduct, 209-214, 216-217
 abstract products, 209-214, 216-217, 222, 231
Accept, 240-251, 254
AcceptImpl, 252, 256, 259
ACE, 309
Acquire, 161-162, 305, 306
Action, 100
Adapter, 284
Add, 278-282, 284, 300
 address-of, 170-171
 after, 16
AFUnit, 223-226
Alexandrescu, Andrei, 29
algorithms
 copy, 40
 compile-time, 76
 linear search, 56
 operating on typelists, 76
Allocate, 83, 85
Allocated, 82
 allocation, small-object. *See also* allocators
 basic description of, 77-96
 default free store allocators and, 78
 fixed-size allocators and, 84-87
 hat trick and, 89-91

 memory chunks and, 81-84
 quick facts, 94-95
 allocators. *See also* allocation, small-object
 default free store, 78
 fixed-size, 84-87
 memory chunks and, 81-84
 workings of, 78-79
allocChunk_, 85, 87
AllowConversion, 192-193
ALU (arithmetic logic unit), 303
AnotherDerived, 197
APIs (application program interfaces), 161, 306
Append, 57-58, 76
Application, 100
 arguments, 114-115, 285-290
 Array policy, 19-20
 arrays, 19-20, 183-184
ArrayStorage, 189-190
assert, 193
AssertCheck, 193
AssertCheckStrict, 193
 assertions, compile-time, 23-26
 associative collections, 203
AssocVector, 210, 277-278
 asynchronous execution, 302
atexit, 134-139, 142-143, 149
ATEXIT_FIXED, 139
AtExitFn, 144-145
AtomicAdd, 305
AtomicAssign, 305
AtomicDecrement, 186
AtomicIncrement, 186
auto_ptr, 108, 159, 170
available_, 79

B

backEnd_, 281, 294
 BackendType, 294
 BadMonster, 219-222
 BadSoldier, 219-222, 230
 BadSuperMonster, 219-221
 BankAccount, 306
 Bar, 139
 Base, 63, 90, 91, 176
 BASE_IF, 37
 BaseLhs, 272, 299
 BaseProductList, 227
 BaseRhs, 272, 284, 299
 BaseSmartPtr, 46
 BaseVisitable, 249, 251, 254
 BaseVisitor, 245, 249-250, 252, 254, 256, 261
 BaseVisitorImpl, 260, 262
 BasicDispatcher, 277-284, 292-294, 297-299
 BasicFastDispatcher, 291-294, 297, 298-299
 before, 16
 BinderFirst, 121
 BindFirst, 121, 127-128
 binding, 119-121, 127-128
 BitBlast, 40-41, 44-46
 blocksAvailable_, 81-83
 blockSize_, 82
 bookkeeping data level, 185-187
 bool, 105, 174-175, 177, 191
 _buffer, 135
 bulk allocation, 86
 butterfly allocation, 86
 Button, 50, 61-62, 219-221

C

callable entities, 103-104
 callbackMap_, 280
 callbacks, 103-104, 205, 280-281
 callbacks_, 205, 281
 CallbackType, 277, 280, 299
 CastingPolicy policy, 288, 299
 CatchAll policy, 260, 262
 Chain, 122, 128
 chaining requests, 122
 char, 35, 38, 114-115
 checking issues, 181-182
 checkingImpl, 193
 Checking policy, 15-16, 188, 193-194
 chunks, of memory, 81-87
 chunkSize, 88

Circle, 203
 class(es). *See also* inheritance; policy classes
 base, 228
 client, 153-154
 decomposing, 19-20
 derived, 228
 final, 29
 generating, with typelists, 64-65
 -level locking operations. 187
 local, 28-29
 object factories and, 200-201
 visitable, 248-255
 visitor, 248-255
 Class, 200
 ClassLevelLockable, 153, 307-309
 Clone, 30, 107, 123, 164, 211-213, 230, 232, 234, 284
 CloneFactory, 214-218
 clone object factories, 211-215
 CLOS, 263
 columns_, 292
 COM (Component Object Model), 133, 192
 command(s). *See also* Command design pattern
 active, 102
 forwarding, 102
 Command, 100
 Command design pattern, 99-104
 basic description of, 100-102
 in the real world, 102-103
 comparison operators, 178-181
 compile-time
 assertions, 23-26
 detecting convertibility and inheritance at, 34-37
 CompileTimeChecker, 25-26
 CompileTimeError, 25
 COMRefCounted, 192
 ConcreteCommand, 100, 102
 ConcreteFact, 232, 234
 ConcreteFactory, 226-228, 230-231, 233-234
 ConcreteLifetimeTracker, 144-145
 concrete products, 222, 227
 const, 44, 114-115, 182-183
 constant(s)
 mapping integral, to types, 29-31
 -time multimethods, 290-293
 ConventionalDialog, 219-221
 conversion
 argument, 114-115, 285-290
 binding as, 119-121

implicit, to raw pointer types, 171-173
 return type, 114-114
 user-defined, 172
 Conversion policy, 36-37, 188, 192-193
 convertibility, detecting, at compile time, 35-37
 copy
 construction, eliding of, 123
 deep, 123, 162-164, 192
 destructive, 168-170
 on write (COW), 165
 Copy, 40, 45
 copy_backward, 144
 copyAlgo, 45
 CORBA (Common Object Request Broker
 Architecture), 115, 133
 counting, reference, 165-167
 covariant return types, 212-213
 Create, 9, 11-12, 14, 31-32, 198, 224, 234
 CreateButton, 50
 CreateDocument, 199
 CreateObject, 208-209
 CreateScrollBar, 50
 CreateShape, 205-206
 CreateShapeCallback, 205
 CreateStatic, 153
 CreateT, 223
 CreateUsingMalloc, 153
 CreateUsingNew, 153
 CreateWindow, 50
 Creation policy, 151, 153
 Creator policy, 7-9, 11-14, 149, 154, 156
 cyclic
 dependencies, 243-248
 references, 168
 cyclicVisitor, 255-257, 261
 Czarnecki, Krzysztof, 54

D

dead reference problem, 135-142
 Deallocate, 82-87
 deallocChunk_, 86-87
 deep copy, 123, 163-164, 192
 DeepCopy, 192
 DEFAULT_CHUNK_SIZE, 93-94, 95
 default free store allocator, 78
 DefaultLifetime, 153
 DEFAULT_THREADING, 94
 #define preprocessor directive, 93, 139
 DEFINE_CYCLIC_VISITABLE, 257
 DEFINE_VISITABLE, 252, 254, 256
 delete, 12, 89-91, 94, 108, 132, 143, 159,
 172-178
 delete[], 184, 189-190
 DeleteChar, 125
 dependency, circular, 141
 DependencyManager, 141
 Deposit, 306
 dereference, checking before, 182
 Derived, 90, 197-198
 DerivedToFront algorithm, 63-65, 76, 272
 design patterns
 Abstract Factory pattern, 69, 49-51, 219-234
 Command pattern, 99-104
 Double-Checked Locking pattern, 146-147,
 149
 Prototype design pattern, 228-233
 Strategy design pattern, 8
 Destroy policy, 20
 destroyed_, 136, 138
 _DestroySingleton, 135
 DestructiveCopy, 192
 destructors, 12-13
 detection, dead-reference, 135-137
 Dialog, 219-221
 Dijkstra, Edgar, 305-306
 DisallowConversion, 192-193
 DispatcherBackend policy, 294
 DispatchRhs, 269-270
 Display, 135-142, 169
 DisplayStatistics, 230
 do-it-all interface, failure of, 4-5
 DocElement, 236-248, 249, 251, 256, 259
 DocElementVisitor, 239-248
 DocElementVisitor.h, 243-244
 DoClone, 213
 DoCreate, 223-224, 227, 232
 DocStats, 236-237, 240-242, 246-247, 248
 Document, 198
 DocumentManager, 198-199
 DottedLine, 212
 double, 306
 Double-Checked Locking pattern, 146-147, 149
 DoubleDispatch, 267, 268
 double dispatcher, logarithmic, 263, 276-285, 297-300
 double switch-on-type, 264-268
 Dr. Dobb's Journal, 125
 Drawing, 201-203
 DrawingDevices, 290

DrawingType, 203
 Dylan, 263
 dynamic_cast, 238, 243, 246, 251, 255, 267,
 285-290, 299
 cost of, 255-256
 DynamicCaster, 288, 289
E
 EasyLevelEnemyFactory, 227, 229, 231
Effective C++ (Meyers), 132
 efficient resource use, 302
 Eisenecker, Ulrich, 54
 ElementAt, 20
 Ellipse, 203, 271, 273
 else, 238
 EmptyType, 39-40, 48, 110
 encapsulation, 99
 EnforceNotNull, 15, 18
 enum, 45
 equality, 173-178
 erase, 278
 Erase, 58-59, 76
 EraseAll, 76, 59
 error(s)
 messages, compile-time assertions and, 24
 reporting, 181-182
 EventHandler, 71
 events, 71, 309
 exceptions, 209
 Execute, 100, 102
 Executor, 269-272
 exists2way, 36
 ExtendedWidget, 18, 164
F
 factorie(s)
 basic description of, 197-218
 classes and, 200-201
 generalization and, 207-210
 implementing, 201-206
 need for, 198-200
 quick facts, 216-217
 templates, 216-218
 type identifiers and, 206-207
 using, with generic components, 215
 Factory, 207-208, 215-217
 FactoryErrorImpl, 209
 FactoryError policy, 208-209, 217-218, 234
 FactoryErrorPolicy, 217-218, 234
 FastWidgetPtr, 17

Field, 67-70, 74-75
 Fire, 275, 270, 271
 firstAvailableBlock_, 81-83
 FixedAllocator, 80-81, 84-85
 FnDispatcher, 280-282, 285, 288, 293-294,
 299-300
 FnDoubleDispatcher, 280
 Foo, 103, 139
 forwarding functions, cost of, 122-124
 free, 143, 189-190
 fun_, 113, 115
 functionality, optional, through incomplete
 instantiation, 13-14
 functions
 forwarding, cost of, 122-124
 static, singletons and, 130
 functor(s)
 argument type conversions and, 114-115
 basic description of, 99-128
 binding and, 119-121
 chaining requests and, 122
 command design pattern and, 100-102
 double dispatch to, 282-285
 generalized, 99-128
 handling, 110-112
 heap allocation and, 124-125
 implementing Undo and Redo with, 125-126
 multimethods and, 282-285
 quick facts, 126-128
 real-world issues and, 122-125
 return type conversions and, 114-115
 Functor1, 106
 Functor2, 106
 FunctorDispatcher, 283-285, 288, 293,
 299-300
 FunctorHandler, 110-112, 117-119, 126
 Functor template, 99-108, 114, 117, 120,
 125-126, 215
 FunctorImpl, 107-112, 123-124, 128, 283, 284
 FunctorType, 284
 FunkyDialog, 219-221

G

GameApp, 230
 Gamma, Ralph, 248
 generalization, 207-210. *See also* generalized
 functors
 generalized functors. *See also* functors
 argument type conversions and, 114-115

- basic description of, 99-128
- binding and, 119-121
- chaining requests and, 122
- command design pattern and, 100-102
- handling, 110-112
- heap allocation and, 124-125
- implementing Undo and Redo with, 125-126
- quick facts, 126-128
- real-world issues and, 122-125
- return type conversions and, 114-115
- GenLinearHierarchy, 71-75, 227, 230, 231
- GenScatterHierarchy, 64-75, 223-226, 227-228
- geronimoswork, 117
- GetClassIndex, 292-293
- GetClassIndexStatic, 292
- GetImpl, 162, 173, 183, 190
- GetImplRef, 162, 190
- GetLongevity, 154
- GetPrototype, 12, 14
- Go, 269, 270, 272, 279
- GoF (Gang of Four) book, 100, 122, 125, 199, 235, 248-249, 255-262
- granular interfaces, 224
- GraphicButton, 61-62
- GUI (graphical user interface), 102

H

- handles, 161
- Harrison, Tim, 146
- Haskell, 263
- hat trick, 89-91
- HatchingDispatcher, 272
- HatchingExecutor, 271
- HatchRectanglePoly, 279
- header files
 - DocElementVisitor.h, 243-244
 - Multimethods.h, 294
 - Typelist.h, 51, 55, 75
 - SmallAlloc.h, 93-95
- heaps, 124-125, 189-190
- HeapStorage, 189-190
- hierarchies
 - linear, 70-74
 - scattered, 64-70
- HTMLDocument, 198

I

- IdentifierType, 209, 213, 215
- IdToProductMap, 214

- #ifdef preprocessor directive, 138-139
- if-else statements, 29, 267, 268, 270
- if statements, 238, 267, 305
- IMPLEMENT_INDEXABLE_CLASS, 293
- implicit conversion, to raw pointer types, 171-173
- IncrementFontSize, 240, 241
- indexed access, 55
- IndexOf, 56, 76, 275
- inequality, 173-178
- inheritance
 - detecting, at compile time, 34-37
 - logarithmic dispatcher and, 279
 - multiple, 5-6
- INHERITS, 37
- Init, 40, 82
- initialization
 - checking, 181-182
 - dynamic, 132
 - lazy, 182
 - object factories and, 197
 - static, 132
- insert, 205
- InsertChar, 120, 122, 125-126
- Instance, 131-132, 135-137, 146, 151
- instantiation, 13-14, 120, 272, 274
- int, 159
- Int2Type, 29-31, 68-69
- interface(s)
 - Abstract Factory design pattern, 223-226
 - application program (APIs), 161, 306
 - do-it-all, failure of, 4-5
 - granular, 224
 - graphical user (GUI), 102
 - separation, 101
- intrusive reference counting, 167. *See also* reference counting
- IntType, 186, 304
- InvocationTraits, 275
- isConst, 47
- isPointer, 47
- isReference, 41, 47
- isStdArith, 47
- isStdFloat, 47
- isStdFundamental, 42-43, 47
- isStdIntegral, 47
- isStdSignedInt, 47
- isStdUnsignedInt, 47
- isVolatile, 47

K

KDL problem, 135-142, 155
 Keyboard, 135-142, 155
 KillPhoenixSingleton, 138
 Knuth, Donald E., 77, 78

L

Lattanzi, Len, 77
 Length, 76
 Less, 180-181
 LhsTypes, 272
 Lifetime policy, 149-153
 LifetimeTracker, 142-143
 LIFO (last in, first out), 134
 Line, 203, 204, 212-213
 linking, reference, 167-168
 LISP, 52
 ListOfTypes, 295
 Lock, 146, 184, 306
 LockedStorage, 189-190
 locking
 class-level, 307
 object-level, 306-308
 pattern, double-checked, 146-147, 149
 semantics, 306-308
 LockingProxy, 184-185
 Log, 135-142
 logarithmic double dispatcher, 263, 276-285, 297-300
 logic_error, 152
 Loki, 70, 77, 91-92, 210, 303, 309
 multimethods and, 263, 268, 277-278, 288, 295-300
 mutexes and, 306
 smart pointers and, 163
 long double data type, 35
 longevity, 139-145, 149, 151
 Lower_bound, 277

M

MacroCommand, 122
 macros, 122, 251-252, 254, 256-257, 261
 "maelstrom effect," 170
 MAKE_VISITABLE, 261
 MakeAdapter, 28-29
 MakeCopy, 164
 MakeT, 223
 malloc, 143
 map, 204-205, 210, 277, 278

mapping
 integral constants to types, 31-32
 type-to-type, 31-33
 Martin, Robert, 245
 maxObjectSize, 88
 MAX_SMALL_OBJECT_SIZE, 93-94, 95
 MemControlBlock, 78-79
 MemFunHandler, 117-119, 126
 memory
 allocators, workings of, 78-80
 chunks of, 81-87
 heaps, 124-125, 189-190
 RMW (read-modify-write) operation, 303-304
 Meyers, Scott, 77, 133-134, 276, 280
 Meyers singleton, 133-134
 ML, 263
 ModalDialog, 27
 Monster, 219-222, 224, 229
More Effective C++ (Meyers), 276, 280
 MostDerived, 63, 76
 multimethods
 arguments and, 285-290
 basic description of, 263-300
 constant-time, 290-293
 double switch-on-type and, 265-268
 logarithmic double dispatcher and, 276-285
 need for, 264-265
 quick facts, 297-300
 symmetry and, 273-274
 Multimethods.h, 294
 MultiThreaded, 6
 MultiThreadedRefCounting, 186-187
 multithreading, 145-148, 302-306
 at the bookkeeping data level, 185-187
 critique of, 302-303
 library, 301-309
 mutexes and, 305-306
 at the pointee object level, 184-185
 reference counting and, 186
 reference tracking and, 186-187
 smart pointers and, 184-187
 mutex_, 146
 mutexes, 146-147, 305-306
 MyController, 27
 MyOnlyPrinter, 130
 MyVisitor, 257

N

name, 206

name cyclic dependency, 243
 new[], 183
 next_, 187
 NiftyContainer, 28-30, 33-34
 NoChecking, 15, 18-19
 NoCopy, 192
 NoDestroy, 153
 NoDuplicates, 60-61, 76
 NonConstType, 47
 nontemplated operators, 176-177
 NonVolatileType, 47
 NoQualifiedType, 47
 NullType, 39-41, 48, 52, 54-56, 62-63, 270

O

object factorie(s)
 basic description of, 197-218
 classes and, 200-201
 generalization and, 207-210
 implementing, 201-206
 need for, 198-200
 quick facts, 216-217
 templates, 216-218
 type identifiers and, 206-207
 using, with generic components, 215
 ObjectLevelLockable, 307-309
 OnDeadReference, 136-139, 150
 OnError, 272
 OnEvent, 70-71
 OnUnknownVisitor, 260, 262
 operators
 operator.*, 104, 116-117
 operator!=, 174, 176, 178
 operator(), 103-113, 116, 122-127, 283, 285
 operator->, 157, 160-162, 165, 182-185
 operator->*, 104, 116-117
 operator*, 157, 161-162, 178-179, 182
 operator<, 144
 operator<=, 178-179
 operator=, 162
 operator==, 176-177, 178
 operator>, 178-179
 operator>=, 178-179
 operator[], 55, 183, 278
 operator T*, 172
 OpNewCreator, 10
 OpNewFactoryUnit, 226, 227, 231, 234
 OrderedTypeInfo, 217, 276-277
 orthogonal policies, 20

OutIt, 40
 ownership-handling strategies, 163-170
 Ownership policy, 183-184, 186, 188, 190-192

P

Paragraph, 237, 241, 247, 249, 254, 256
 ParagraphVisitor, 246, 247, 248, 251
 parameter(s)
 template, 10-11, 64, 105
 types, optimized, 43-44
 ParameterType, 43-44, 47, 123
 ParentFunctor, 111, 120-121
 Parm1, 111
 Parm2, 111
 ParmN, 109
 Parrot, 117-119
 pattern(s)
 Abstract Factory pattern, 69, 49-51, 219-234
 Command pattern, 99-104
 Double-Checked Locking pattern, 146-147, 149
 Prototype design pattern, 228-233
 Strategy design pattern, 8
Pattern Hatching (Vlissides), 133
 pData_, 86
 pDocElem, 246
 pDuplicateShape, 212
 pDynObject, 140
 pFactory_, 222
 Phoenix Singleton, 137-142, 149, 153
 pimpl idiom, 78
 pInstance_, 131-132, 136, 138, 146-148, 151
 placement new, 138
 pLastAlloc_, 89
 POD (plain old data) structure, 45-46
 Point3D, 70
 pointee_, 160, 161, 178, 183
 PointeeType, 41-42, 47, 160-161
 pointee object level, 184-185
 pointer(s)
 address-of operator and, 170-171
 arrays and, 183-184
 basic description of, 157-195
 checking issues and, 181-182
 copy on write (COW) and, 165
 deep copy and, 163-164
 destructive copy and, 168-170
 equality and, 173-178
 error reporting and, 181-182

- failure of the do-it-all interface and, 5
 - handling, to member functions, 115-119
 - implicit conversion and, 171-173
 - inequality and, 173-178
 - multithreading and, 184-187
 - ordering comparison operators and, 178-181
 - ownership-handling strategies, 163-170
 - quick facts, 194-195
 - raw, 171-173
 - reference counting and, 165-167, 186
 - reference linking and, 166-168
 - reference tracking and, 186-187
 - traits of, implementing, 41-42
 - types, implicit conversion, 171-173
 - `PointerToObj`, 118
 - `PointerTraits`, 41-42
 - `PointerType`, 160, 190-191
 - policies. *See also* policy classes
 - basic description of, 3, 7-11
 - `BasicDispatcher` and, 293-294
 - `BasicFastDispatcher` and, 293-294
 - compatible, 17-18
 - decomposing classes in, 19-20
 - enriched, 12
 - multimethods and, 293-294
 - noncompatible, 17-18
 - orthogonal, 20
 - singletons and, 149-150, 152-153
 - stock, 152-153
 - policies (listed by name). *See also* policies
 - Array policy, 19-20
 - `CastingPolicy` policy, 288, 299
 - `CatchAll` Policy, 260, 262
 - Checking policy, 15-16, 188, 193-194
 - Conversion policy, 36-37, 188, 192-193
 - Creation policy, 151, 153
 - Creator policy, 8-9, 11-14, 149, 155, 156
 - Destroy policy, 20
 - `DispatcherBackend` policy, 294
 - `FactoryError` policy, 208-209, 217-218, 234
 - `Lifetime` policy, 149-153
 - Ownership policy, 183-184, 186, 188, 190-192
 - Storage policy, 17, 185, 188, 189-190
 - Structure policy, 16-17
 - `ThreadingModel` policy, 16, 149, 151-153, 186, 303-309
 - policy classes. *See also* classes
 - basic description of, 3-22
 - combining, 14-16
 - customizing structure with, 16-17
 - destructors of, 12-13
 - implementing, 10-11
 - `poly`, 271, 279
 - `Polygon`, 203, 212
 - polymorphism, 78, 163-164
 - Abstract Factory implementation and, 228-229
 - multimethods and, 264
 - object factories and, 197-200, 211-212
 - `prev_`, 187
 - `Printer`, 161-162
 - `printf`, 105
 - `printingPort_`, 130
 - `priority_queue`, 142-145
 - `ProductCreator`, 210-211, 216-217
 - Prototype design pattern, 228-233
 - `PrototypeFactoryUnit`, 231-232, 234
 - prototypes, 228-233
 - `pTrackerArray`, 143
- ## Q
- qualifiers, stripping, 44
- ## R
- `RasterBitmap`, 237, 241, 247, 249, 256
 - `RasterBitmapVisitor`, 247, 251
 - `realloc`, 143
 - `Receiver`, 100
 - `Rectangle`, 267, 279, 289
 - `RectanglePoly`, 279
 - `Redo`, 125-126
 - `RefCounted`, 6, 192
 - `RefCountedMT`, 192
 - reference(s)
 - counting, 165-167, 186
 - linking, 167-168, 186-187
 - `ReferencedType`, 41-42, 47
 - `Reflinked`, 192
 - `RegisterShape`, 206
 - `RejectNull`, 194
 - `RejectNullStatic`, 193-194
 - `RejectNullStrict`, 194
 - `Release`, 161-162, 192, 305, 306
 - `Replace`, 76
 - `ReplaceAll`, 61, 76
 - `Reset`, 162
 - resource leaks, 133
 - `Result`, 55
 - `ResultType`, 111, 269, 284

return type(s)
 conversion, 114-115
 covariant, 228
 generalized functors and, 114-115
 RISC processors, 148
 RMW (read-modify-write) operation, 303-304
 RoundedRectangle, 272, 286-287, 289
 RoundedShape, 286-287, 289
 runtime_error, 137, 209-210, 300
 runtime type information (RTTI), 215, 245, 255, 276

S

safe_reinterpret_cast, 26
 safewidgetPtr, 17
 sameType, 36
 Save, 201-203
 scanf, 105
 ScheduleDestruction, 149-150
 Schmidt, Douglas, 146-147
 Scroll, 122
 ScrollBar, 50, 61-62
 Secretary, 5
 Section, 254
 Select, 33-34, 62
 semantics
 failure of the do-it-all interface and, 4-5
 functors and, 99
 locking, 306-308
 semaphores, 309
 SetLongevity, 140-145, 149
 SetPrototype, 12, 14, 232
 Shape, 201-202, 265, 268, 279, 286-289
 ShapeCast, 289
 ShapeFactory, 204-205, 215
 Shapes, 290
 SillyMonster, 219-222, 226
 SillySoldier, 219-222, 230
 SillySuperMonster, 219-222
 SingleThreaded, 153, 308
 singleton(s). *See also* SingletonHolder
 basic C++ idioms supporting, 131-132
 dead reference problem and, 135-142
 destroying, 133-135
 double-checked locking pattern and, 146-147
 failure of the do-it-all interface and, 4-5
 implementing, 129-154
 longevity and, 139-145
 Meyers singleton, 133-134
 multithreading and, 145-148
 Phoenix, 137-142, 149, 153
 static functions and, 130
 uniqueness of, enforcing, 132-133
 SingletonHolder, 3, 91, 129-130, 148-153, 215.
 See also singletons
 assembling, 150-152
 decomposing, 149-150
 quick facts, 155-156
 working with, 153-156
 SingletonWithLongevity, 153, 154
 sizeof, 25, 34-35, 82, 90-91
 size_t, 36, 79
 skinnable programs, 103
 SmallAlloc.h, 93-95
 small-object allocation
 basic description of, 77-96
 default free store allocators and, 78
 fixed-size allocators and, 84-87
 hat trick and, 89-91
 memory chunks and, 81-84
 quick facts, 94-95
 SmallObjAllocator, 80-81, 87-89, 92-95
 SmallObject, 80-81, 89, 91-95, 123-124
 smart pointer(s)
 address-of operator and, 170-171
 arrays and, 183-184
 basic description of, 157-195
 checking issues and, 181-182
 copy on write (COW) and, 165
 deep copy and, 163-164
 destructive copy and, 168-170
 equality and, 173-178
 error reporting and, 181-182
 failure of the do-it-all interface and, 5
 implicit conversion and, 171-173
 inequality and, 173-178
 multithreading and, 184-187
 ordering comparison operators and, 178-181
 ownership-handling strategies, 163-170
 quick facts, 194-195
 raw, 171-173
 reference counting and, 165-167, 186
 reference linking and, 167-168
 reference tracking and, 186-187
 types, implicit conversion, 171-173
 SmartPtr, 3, 6-7, 14-19, 44, 87, 157-195
 Soldier, 219-222, 224, 229-230
 SomeLhs, 278, 284

SomeRhs, 278, 284
 SomeThreadingModel, 309
 SomeVisitor, 250, 254
 spImpl_, 112
 statements
 if, 238, 267, 305
 if-else, 29, 267, 268, 270
 static, 280
 static_cast, 114, 285-290, 299
 STATIC_CHECK, 25
 StaticDispatcher, 268-276, 297-298
 static manipulation, 6
 Statistics, 249
 stats, 246-247
 STL, 115, 171
 StorageImpl, 189, 190
 Storage policy, 17, 185, 188, 189-190
 Strategy design pattern, 8
 String, 302-303, 307
 Stroustrup, Bjarne, 202
 struct, 45
 structure(s)
 customizing, with policy classes, 16-17
 POD (plain old data), 45-46
 specialization of, 7
 structure policy, 16-17
 SuperMonster, 219-222, 229
 SUPERSUBCLASS, 37, 62
 Surprises.cpp, 224
 Sutter, Herb, 77
 switch, 203
 SwitchPrototype, 13-14
 symmetry, 273-274
 synchronization objects, 303

T

template(s)
 advantages of, 6-7
 implementing policy classes with, 11
 skeleton, Functor class, 104-108
 specialization, partial, 53-54
 template parameters, 10-11, 64, 105
 templated operators, 176-177
 TemporarySecretary, 5
 Tester, 178
 Tester*, 178
 TestFunction, 113-115
 TextDocument, 198
 ThreadingModel policy, 16, 149, 151-153, 186, 303-309
 time
 separation, 101
 slicing, 201
 TList, 53-65, 75, 120, 127, 223, 231-234, 261
 Tools menu, 102
 trampoline functions (thunks), 280-283
 Trampoline, 281
 translation units, 132
 Triangle, 289
 tuples, generating, 70
 type(s)
 atomic operations on, 303-304
 conversions, generalized functors and, 114-115
 detection of fundamental, 42-43
 identifiers, 201, 206-207
 integral, 303-305
 modifiers, 308-309
 multiple inheritance and, 6
 multithreading and, 303-305
 parameters, optimized, 43-44
 replacing an element in, 60-61
 safety, loss of static, 4
 selection, 33-34
 -to-type mapping, 31-33
 traits, 38-46
 Type2Type, 32-33, 223
 TypeAt, 55, 76
 TypeAtNonStrict, 55, 76, 109
 typedef, 14, 19, 51, 54, 215, 295
 typeid, 38, 267
 type_info, 37-39, 54, 206-207, 213-215, 276-277, 295
 TypeInfo, 38-39, 48
 TYPelist, 295
 Typelist.h, 51, 55, 75
 typelists, 223, 268, 272
 appending to, 57-58
 basic description of, 49-76
 calculating length and, 53-54
 class generation with, 64-75
 compile-time algorithms operating on, 76
 creating, 52-53
 defining, 51-52
 detecting fundamental types and, 42-43
 need for, 49-51
 partially ordering, 61-64
 quick facts, 75

searching, 56-57
TypesLhs, 269-270, 274
TypesRhs, 269, 270, 274
TypeTraits, 41-48, 123, 183

U

UCHAR_MAX, 83
Undo, 125-126
unique bouncing virtual functions, 238
Unlock, 184
UpdateStats, 237-238
upper_bound, 144
use_size, 91

V

ValueType, 33
vector, 183
VectorGraphic, 244
VectorizedDrawing, 259
Veldhuizen, Todd, 54
virtual
 constructor dilemma, 229
 functions, 238
visit, 249-250, 254, 256
visitable, 249, 251, 252-254
visitor design pattern
 acyclic, 243-248

 basic description of, 235-262
 catch-all function and, 242-243, 258-259
 cyclic, 243-248, 254-257
 generic implementation of, 248-255
 hooking variations, 258-260
 nonstrict visitation and, 261
 quick facts, 261-262
visitParagraph, 240, 242
visitRasterBitmap, 242
visitVectorGraphic, 244
Vlissides, John, 133
void*, 172
volatile, 151, 308-309, 308-309
volatileType, 151, 309

W

widget, 26-32, 38, 44, 61-62, 159, 164, 184, 309
widgetEventHandler, 71-74
widgetFactory, 49-51
widgetInfo, 65-67
widgetManager, 9-14, 19-20
window, 50
withdrawal, 306

X

X Windows, 103